TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Sander Puntso 176824IAPM

REST PROTOKOLLIL PÕHINEVATE TARKVARATEENUSTE ANDMETEGA JUHITAVAKS MUUTMINE KONKREETSE TEENUSE NÄITEL

Magistritöö

Juhendaja: Erki Eessaar
PhD

Tallinn 2020

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: Sander Puntso

20.05.2020

Annotatsioon

Käesoleva magistritöö üheks eesmärgiks on kavandada ja realiseerida REST protokollil põhinev andmetega juhitav rakendusliides (API), mis võimaldab sisendandmetega juhtida teenuse äriloogika operatsioonide käitumist. Töö käigus arendatav näidisrakendus on jaotustarkvara regulatsiooni funktsionaalsust esitav alamsüsteem. Arendatav rakendusliides on näidisprojekt. Töö teiseks eesmärgiks on koostada selle põhjal disainimustri formaadis juhend REST protokollil põhinevate ja C# programmeerimiskeeles kirjutatud rakendusliideste arendamiseks.

Töö tulemusena valmis C# programmeerimiskeeles kirjutatud REST protokollil põhinev regulatsiooni rakendusliides. Valminud tarkvara arendati kahes iteratsioonis. Esimeses iteratsioonis valmis püstitatud nõuetele vastav tarkvara, mille regulatsiooni protsessi äriloogika on koodis defineeritud. Teises iteratsioonis valmisid täiendused olemasolevale tarkvarale, mis võimaldavad sisendandmetega muuta kirjeldatud regulatsiooni tootekoguste arvutamise loogikat. Valminud täiendused on API sisendi jaoks JSON formaati muundatud C# avaldised ja avaldistepuud, mis rakenduse äriloogika kihis muundatakse tagasi C# avaldisteks ning käivitatakse.

Näidistarkvara andmetega juhitavaks muutmise sammude põhjal kirjeldati töö viimases osas disainimustri formaadis juhend. Juhendis kirjeldatud sammudega on võimalik läbi teha uus arendusprotsess, mille tulemuseks on näiteprogrammile sarnane andmetega juhitav C# keeles kirjutatud REST API.

Lõputöö on kirjutatud eesti keeles keeles ning sisaldab teksti 74 leheküljel, 6 peatükki, 53 joonist, 8 tabelit.

Abstract

Changing REST Protocol-based Software Services to Data-driven in the Example of a Service

The first goal of this master's thesis is to design and develop a REST protocol-based data-driven API. The data-driven API accepts structured input data that represents business logic decisions and provides a possibility for the interface users to alter the functionality of the software. The API is an example project that is used as the basis to achieve the second goal of the thesis. The second goal of this thesis is to write down, in the form of a design pattern, a guideline for creating data-driven REST protocol-based APIs that are written in C# programming language.

In the first half of the thesis, the goal is to research data-driven design and related work in the field and to design and develop a working data-driven REST API. In the first iteration of this development process, a magazine distribution regulation software is designed and developed as a REST service. The developments in the first iteration consist only of the main functionality that is described in use cases.

After completing development of the designed software it is altered in a way that it accepts expressions in JSON format as an input to alter the functionality in the software's business logic layer. The business logic function deserializes input data to C# programming language *Expression* base class, then compiles and executes it. The results of the development are validated based on altering the product count calculation logic with API input.

Data-driven REST API development in C# is described by using a design pattern format in the second part of the thesis. With the steps presented in this specification, it is possible to recreate a similar data-driven REST API software solution to what was developed as an example software in this thesis. The proposed solution is not a pattern but rather a pattern candidate because there are not three examples of its usage.

Evaluating the provided example solution and the described development process in a design pattern format have brought up some drawbacks of the solution. Even though the provided data-driven solution itself is working functionality-wise, it is not as easy and comfortable to use as a regular API. Developers of tenants' systems might use Visual Studio environment for writing expressions and serializing these to JSON format. That process expects that these developers are familiar with the systems' database schema in order to use the data-driven API to its' full extent. A possible further development of this work would be to create a graphic user interface for constructing expression trees. This interface would be a "sandbox" environment that would help with integrating the data-driven API.

If the underlying software is developed in a way that the API tenants use a shared database, then executing these expressions might provide a possibility for one tenant to access another tenants' data via shared entities, for example through a classifier table (reference data). Thus, the data of tenants should be separated. To achieve this, the service provider could use the database-per-tenant solution. In this case the software can only access the specific tenant's database without knowing any connection arguments for the other tenants' databases. Another solution is to use row-level-securtity features of database management systems that allow controlling access to rows of tables.

In addition, when developing such systems, one has to take into account that the response time for each request might considerably lengthen when using complex expressions as an input for the API. In this thesis load testing on two REST APIs was conducted, one of which was data-driven and the other not. Based on the results of the load tests, it was clear that with increasing load on the system, the endurance limit of the data-driven API was lower. Deserializing and executing expressions takes longer than it would take for the compiled code to execute the same functionality.

The thesis is in Estonian and contains 74 pages of text, 6 chapters, 53 figures, 8 tables.

Lühendite ja mõistete sõnastik

API	(*application program interface*) rakendusliides, tarkvara liides
Boolean muutuja	kahendmuutuja ehk muutuja, mille väärtused saavad olla vaid tõene või väär
Disainiteadus	(*design science*) infotehnoloogia uurimismetoodika, kus uurimistöö ja arenduse tulemusena esitletakse uudset lahendust või tehnoloogiat
HTTP	(*Hypertext Transfer Protocol*) andmevahetusprotokoll veebis ja arvutivõrkudes
JSON	(*Javascript Object Notation*) Javascript programmeerimiskeele andmevorming
LINQ	(*Language Integrated Query*) .NET (sh C#) programmeerimiskeele laiendus andmebaasist päringute tegemiseks
ORM	(*Object-Relational Mapping*) objekt-relatsioonvastendus – tehnika, mille abil saab objektorienteeritud programmeerimiskeeles kirjutatud programmi panna kasutama relatsioonilises andmebaasis olevaid andmeid
REST	(R*epresentational State Transfer*) veebiteenuste tarkvara arhitektuuri liidese laad; teenuse andmevahetusliidese esitlusstiil
Rutiinne kavandamine	(*routine design*) uurimismetoodika, kus probleemi lahendatakse kasutades olemasolevaid töötavaid lahendusi

SaaS	(*Software as a Service*) tarkvarateenuste litsentseerimis- ja jaotusmudel – tarkvara või tarkvara liidesed on keskselt majutatud ja klientidele tellimuspõhiselt jaotatud
SQL	(*Structured Query Language*) struktureeritud andmebaasikeel, mis realiseerib relatsioonilist andmemudelit ning milles on muuhulgas laused andmebaasiobjektide haldamiseks, andmete käitlemiseks, tehingute juhtimiseks ja õiguste haldamiseks.
Vooskeem	(*flow diagram*) töövoo samm-sammuline graafiline esitlusdiagramm

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Aja jooksul tarkvara muutmine ärikeskkonna, seadusandluse, tehniliste võimaluste, vajaduste ning arusaamade muutumise tõttu, ehk tarkvara evolutsioon, on tarkvara elutsükli loomulik ja paratamatu osa. Tarkvara teenusena (*Software as a Service*, edaspidi SaaS) pakkumine on pilvandmetöötluse mudel, kus andmete töötlust ja hoiustamist pakutakse ühiskasutatava arvutusressursside kogumina. Need teenused on kasutajale kättesaadavad rakendusliidesete (*application programming interface*, edaspidi API) või rakenduste kaudu. Pakutavate ressursside haldamist ja kohandamist teostab teenusepakkuja, st teenusepakkuja kohustuseks on tagada tarkvara muutmine selle elutsükli vältel [1].

Tarkvara andmetega juhitavaks muutmine tähendab, et tarkvara arenemise võimalus ja võimekus on tarkvarasse sisse ehitatud ning tarkvara käitumise muutmiseks ei pea muutma tarkvara lähtekoodi. Selle asemel tuleb muuta andmeid, mida tarkvara töö käigus loeb ja mis määravad ära tarkvara käitumise. Tarkvara tegijalt eeldab see tarkvara arendusprotsessi jooksul täiendavat pingutust.

SaaS teenuste puhul kasutavad erinevad kliendid ja klientsüsteemid ühiselt samu ressursse – jagatud andmebaasi, programmi, teenust. Ühisvalduslik arhitektuur (*multi-tenant architecture*) tähendab SaaS mudeli mõistes, et iga valdaja „osa" süsteemist on eraldatud teistest osadest. Valdaja on grupp kasutajaid, kes jagavad süsteemis sama vaadet, mis hõlmab ligipääsetavaid andmeid, sätteid ning funktsionaalsust. Ühisvaldusliku arhitektuuri alusel loodud süsteemis andmete hoiustamiseks on erinevaid disainilahendusi. Valdajapõhise andmebaasi disaini korral on iga valdaja jaoks loodud oma andmebaas. Valdajad jagavad sellisel juhul tarkvara ning andmebaasi disaini. Jagatud andmebaasi disaini puhul on kõigi valdajate andmed ühise andmebaasi samades andmestruktuurides ning andmete eraldamiseks lisatakse kirjetele valdaja identifikaator [2].

Rakendusliidese-põhiste (API-põhiste) SaaS teenuste valdajateks on nende liideste teenuseid kasutavad tarkvarasüsteemid ning nende süsteemide kasutajad. SaaS API puhul

saab otsesteks kasutajateks pidada tarkvaraarendajaid, kes integreerivad pakutavaid liideseid arendatavatesse süsteemidesse. Selliste teenuste puhul on oluline, et klientidel ja klientsüsteemidel oleks võimalikult suurel määral võimalik teenuseid vastavalt oma vajadustele kohandada [3]. Paraku ei ole teenuseid kasutavatel arendajatel enamasti võimalust muuta liideste realiseeritud äriloogikat. Seega on funktsionaalsete nõuete muutumisel vaja teenuse haldajal täiendada olemasolevaid teenuseoperatsioone või luua uusi. Teenuse andmetega juhtimise võimaluse võib anda teenuse haldajale, kuid see eeldab ikkagi temalt pidevat valmisolekut reageerida teenuse kasutajatelt tulevatele muutmise soovidele. Kuidas saaks seda koormust vähendada ja teenuse kasutajate jaoks paindlikkust suurendada?

Teenuse andmetega juhtimise võimaluse andmine teenuse kasutajatele tähendab, et liidese sisendparameetrite väärtustamise kaudu on teenuse kasutajal võimalik defineerida avaldised või reeglid, mille alusel programmi käivitamise hetkel tehakse otsuseid. Sellisel juhul eeldab funktsiooni käitumises muudatuste tegemine operatsioonide sisend- ja väljundparameetrite tundmist, sh arusaamist nende võimalikest väärtustest ja seostest. Sellise API eeliseks on paindlikkus, kuid probleemiks on suurenenud keerukus ja vähenenud mugavus API kasutaja jaoks. Samuti iseloomustaksid sellist API-t mitmed lähtekoodi “halvad lõhnad” [4] [5], mis kaudselt osutavad ka sellele, et API arendajal tuleb selle loomiseks teha tavalisest rohkem tööd.

Tarkvara teenusena printsiibi põhiolemusest tuleneb ka antud töö aluseks olev põhiprobleem – teenusel on palju erinevaid valdajaid ja seega kasutatakse teenusepakkuja teenuseid (rakendusliidese operatsioone) erinevates klientsüsteemides. Kliendi jaoks on tarkvara valikul oluliseks kriteeriumiks, kas pakutav funktsionaalsus rahuldab tema (organisatsiooni) jaoks püstitatud nõudeid. Erinevatel klientidel on erinevad funktsionaalsed nõuded ning on oluline, et klient saaks ka ise teenuseid vastavalt vajadusele kohandada ilma, et peaks ootama teenusepakkuja poolseid muudatusi. Teenusepakkuja poolelt on optimaalne, kui klientide muutuvad nõudmised süsteemile on võimalikult suurel määral lahendatavad kliendipoolsete tegevuste läbi, vajaduseta muuta tarkvara lähtekoodi. Kui ühiskasutatava teenuse lähtekood muutub, siis tähendab see muudatust kõikidele muutunud funktsionaalsust kasutavatele klientidele. Olenevalt muudatusest võib see tähendada kliendile lisatööd teenust integreerivate süsteemide arendamisel/täiendamisel [6]. Käesolev töö otsib võimalust, kuidas saaks süsteemi käitumist muuta nii, et sellist liiki lisatööd ei peaks tegema.

1.1 Taust ja probleem

Vajadus tarkvara andmetega juhitavuse uurimiseks tuleneb tõsiasjast, et tarkvara ei ole peale valmimist kivisse raiutud, vaid peab kogu oma eluea jooksul muutuma ehk evolutsioneeruma. Tarkvara täiendamise ja lisafunktsionaalsuse arendamisega kasvab tarkvara keerukus ning haldamiseks vajalik pingutus. Küll aga on tarkvara evolutsioon hädavajalik ning ühel või teisel viisil peab tarkvara täitma püstitatud ning ajas muutuvad nõuded [7]. Tarkvara evolutsiooni oluliseks osaks on tarkvara muudatused ning nende elluviimine tarkvara tarneahelas [8]. Muudatused tarkvaras on väga veaohtlikud ning empiirilised uuringud on näidanud, et tarkvara hoolduse käigus osutuvad 15–70% parandustest esimese väljalaske ajal vigasteks [9] [10].

Probleemipüstitus põhineb Ken Downsi ajaveebi postitusele [11], kus kirjeldatakse andmetega juhitavuse printsiipi ajakirjade jaotusega tegeleva organisatsiooni regulatsiooniprotsessi näitel. Downsi esitletava probleemi aluseks on kehvasti disainitud tarkvarast tulenev olukord, kus tarkvara haldaja ja omanik ei julge teha tarkvaras muudatusi, sest nad kardavad selle tegevusega tarkvara töös vigu tekitada. Kuna kirjeldatud süsteem ei ole arendatud andmetega juhitavaks, tähendab muudatus igal juhul täiendusi programmikoodis. Selliselt kirjeldatud situatsioonis on kliendil valikuks kas leppida olemasoleva funktsionaalsusega seni, kuni leitakse sobivam alternatiiv või teha lisapingutusi ja ka lisakulutusi muutunud nõuetele mittevastava programmi täiendamisel. Olemasoleva olukorraga leppimine või muudatuse tegemise venimine tähendab, et klient ei saa oma äri muuta nii nagu talle on parajasti vaja ning tarkvara kirjutab ette, kuidas äri ajada, mitte ei ole äri toetavas rollis, milles see peaks olema.

Käesoleva töö eesmärgiks on kavandada ja C# keeles realiseerida REST protokollil põhinev andmetega juhitav regulatsiooniprogrammi API, mis võimaldab rakendusliidest integreerival tarkvaraarendajal juhtida sisendandmetega operatsioonide käitumist. Töö käigus arendatav rakendusliides on näidisprojekt, mille põhjal koostatakse juhend andmetega juhitavate C# keeles kirjutatud REST protokollil põhinevate rakendusliideste arendamiseks. See juhend pannakse kirja kasutades disainimustrite formaati. Koostatud juhend annab üldise ülevaate andmetega juhtimise vajadusest ning kirjeldab tegevused olemasoleva või uue tarkvara muutmiseks andmetega juhitavaks läbi parameetrite väärtuste ja konfiguratsiooni.

Töö näidisprojekt arendatakse C# programmeerimiskeeles, kasutades andmepäringute tegemiseks C# keele põhist integreeritud päringu komponenti (*language integrated query*, edaspidi LINQ) ja C# avaldisi [12] [13].

1.2 Ülevaade tööst

Lõputöö sisuline osa on jagatud kolmeks eraldiseisvaks osaks.

Peatükis 2 antakse ülevaade andmetega juhitavuse printsiipidest. Samuti kirjeldatakse ülevaatlikult, millised on selles valdkonnas antud hetkel olemasolevad lahendused ja tehnoloogilised võimalused.

Töö teises osas, peatükkides 3 ja 4, kirjeldatakse näidisprogrammi nõuded lähtudes eelkõige Ken Donwsi [11] ajaveebi postituses kirjeldatud jaotussüsteemi regulatsiooniprotsessist. Postituses toodud üldise kirjelduse alusel (postituses ei ole koodinäiteid ega süsteemi detailset kavandit) defineeritakse võimaliku sarnase süsteemi (jaotustarkvara regulatsiooni alamsüsteemi) nõuded ning realiseeritakse see REST teenusena. Loodud tarkvaralahendus kohandatakse andmetega juhitavaks REST päringute sisendandmete kaudu, lähtuvalt peamisest regulatsiooni loomise operatsioonist.

Töö kolmandas osas esitatakse disainimustri formaati järgiv juhend, mille alusel on võimalik arendada ja täiendada olemasolevaid REST protokollil põhinevaid liideseid andmetega juhitavuse põhimõtetele vastavaks. Juhend koostatakse eelnevas peatükis kirjeldatud sammude põhjal ning kajastab ka kirjeldatud protsessi puuduseid.

1.3 Metoodika ja tulemuste valideerimine

Antud magistritöö uurimismetoodikaks on valitud disainiteadus (*design science*) [14]. Disainiteaduse metoodika kasutamise tulemuseks on tehniline tehis, mis põhineb oma valdkonnas juba kogutud teadmistel ning mille headust on peale valmimist mingil viisil kontrollitud.

Antud töö tulemuseks on kaks tehist. Esimeseks tehiseks on konkreetne andmetega juhitav tarkvarateenus ning teiseks tehiseks on selliste teenuste loomise üldine mustripõhine juhend. Antud uurimismetoodika sammud selle töö puhul on uue tarkvara kavandamine, arendamine, hindamine, tehtud tegevuste üldistamine ning üldistuse

tulemuste disainimustrite formaadis kirjeldamine. Kavandamine kujutab endast probleemi hindamist ja lahenduse pakkumist, arendamine tähendab pakutud lahenduse realiseerimist ning hindamine tähendab loodud lahenduse nõuetele vastavuse kontrollimist ja lõppjärelduste tegemist.

Metoodikat võib pidada sobivaks, kuna tegu on uudsete tehiste kavandamise ja arendamisega. Tegemist ei ole rutiinse kavandamisega (*routine design*), st loodav tehis ei põhine ainult olemasolevatele tehnikatele ja praktikatele [15].

Antud töö tulemuste valideerimiseks hinnatakse, kas töö praktilises etapis arendatud näidistarkvara funktsionaalsus vastab esialgsetele funktsionaalsetele nõuetele ja on liidese sisendite kaudu andmetega muudetav. Andmetega juhitavaks võib tarkvara pidada siis, kui muutunud nõuded tarkvara funktsionaalsusele on võimalik rahuldada andmete muutmise kaudu, ilma, et oleks vaja teha täiendusi lähtekoodis. Seejuures on oluline, et tarkvara andmetega juhitavus ei mõjutaks rakenduse põhifunktsionaalsust.

Töö viimases etapis valminud disainimustrite formaadis juhendi valideerimiseks on võimalik kirjeldatud sammudega läbi teha uus arendusprotsess, mille tulemuseks oleks andmetega juhitav REST API. Antud töös seda läbi ei tehta, st veel mingeid uusi rakendusliideseid andmetega juhitavaks ei muudeta.

2 Taust

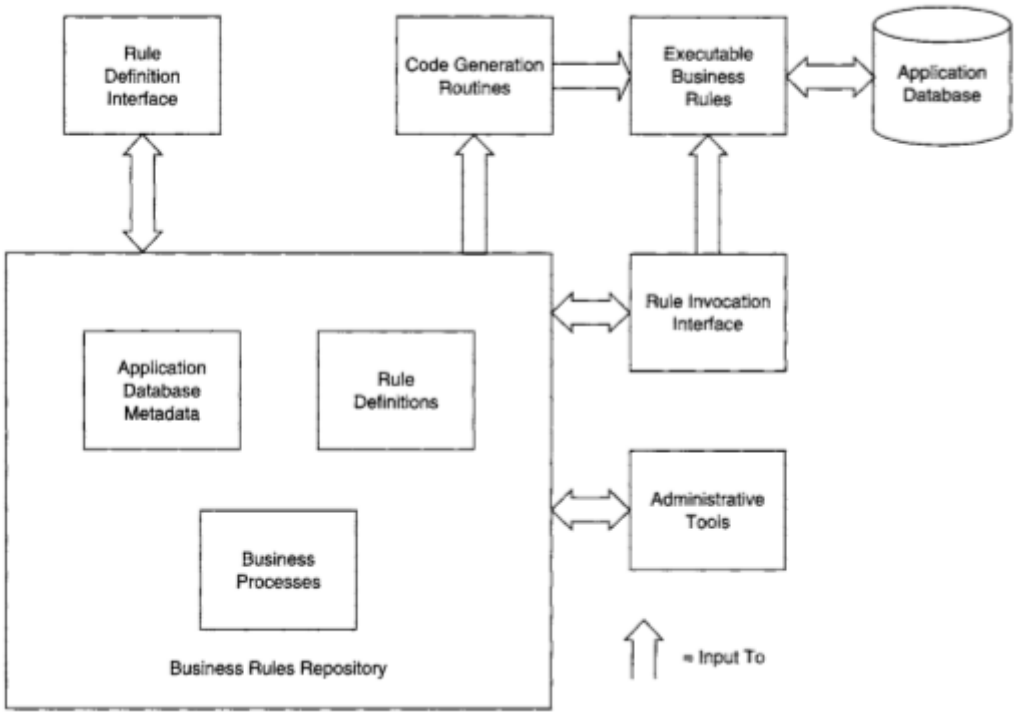
Käesolevas peatükis antakse ülevaade andmetega juhitavusega seotud põhimõtetest ja lahendustest. Tutvustatakse tehnilisi lahendusi, mis otseselt või kaudselt on seotud töö eesmärgiga või realiseeritava tehisega.

Andmetega juhitava tarkvara mõiste ei ole lõputöö teema püstituses kirjeldatud moel eriti levinud praktika ja seda mitmetel põhjustel. Andmetega juhitava tarkvara riskideks on madal kvaliteet, suurenenud veaoht ja skaleeritavuse vähenemine. Need on riskifaktoriks nii tarkvara pakkuja kui ka kliendi jaoks [16]. Lisaks on nõndanimetatud "üks suurus sobib kõigile" (*one size fits all*) lahendused disainitud nii, et kohandamist on võimalik teha vaid piiratud ulatuses, vältimaks võimalikke probleeme tarkvara töös. Piiratud kliendipoolset tarkvara kohandamist õigustab peamiselt ka asjaolu, et tarkvaraarendaja poolt tehtud muudatused on reeglina töökindlamad võrreldes kliendipoolsete muudatustega [17].

2.1 Ärireeglite mootorid

Tarkvara käivitamisel tehtavad otsused realiseerivad ärireegleid, mis on oma olemuselt definitsioon või kitsendus mingist ärilisest omadusest, mille rakendamise tulemusena muudetakse või kontrollitakse mõnda ärilise operatsiooni käitumist [18]. Need reeglid tuleb tarkvara nõuete kogumise käigus üles leida. Ärireeglite haldussüsteem (*Business Rules Management Systems*, BRMS) on tarkvarasüsteem, mille kaudu on võimalik kirjeldada, testida, käivitada ja hallata programmis tehtavaid otsuseid. Selle abil rakendatavad ärireeglid on esitatud mingil viisil andmetena, mitte osana programmikoodist [19]. Mõte on selles, et ärireeglid ei oleks sissekodeeritud rakenduse äriloogika koodi ja andmebaasis defineeritud kitsendustesse, vaid selle asemel on need tõstetud „sulgude ette", eraldi süsteemi, kus neid saab hallata ja kust neid programmide täitmise käigus loetakse ja rakendatakse (nii nagu andmetele juurdepääsu tagamise ja andmete haldamise vastutus on eraldi tarkvara – andmebaasisüsteemi – ülesanne). Eesmärk on, et ärireegleid saaksid muuta lõppkasutajad, ilma arendajate abita [20].

Ärireeglite haldussüsteemi keskmeks on ärireeglite hoidla (*Business Rules Repository*) ning seda reeglite rakendamiseks kasutav ärireeglite mootor (*Business Rules Engine*). Chisholm [18] kirjeldab, kuidas ärireeglite hoidla võib olla realiseeritud SQL-andmebaasina, kus on hulk tabeleid, milles on defineeritud rakenduse andmemudel, ärireeglid ja rakenduse teostatavad äriprotsessid. Graafilise ärireeglite defineerimise liidese (*Rule Definition Interface*) kaudu on kasutajatel võimalik kirjeldada ärireeglid, mis salvestatakse ärireeglite hoidlasse. Salvestatud ärireegleid kasutatakse koodi genereerimise protsessides (*Code Generation Routines*), kus luuakse käivitatavad ärireeglid (*Executable Business Rules*). Nende reeglite käivitamine toimub läbi reeglite aktiveerimise liidese (*Rule Invocation Interface*) või läbi süsteemi enda automaatsete protsesside. Reeglite käivitamise tulemusel loetakse või uuendatakse kirjeid andmebaasis. Administratiivsete tööriistade (*Administrative Tools*) kaudu on võimalik kontrollida reeglite käivitamist ning luua raporteid käivitatud reeglite kohta.



Joonis 1: Ärireegli mootori komponentide ülevaade [18]

Koodi genereerimise protsessides on võimalik eristada kolme erinevat lähenemist [18].

- Kasutaja defineerib reegli käivitataval kujul. Selliselt on tarkvara sisendiks programmikood või muu programmikoodis käivitatav sisend, mille käivitamine nõuab vähest või üldse mitte teisendamist.

- Reeglite mootor salvestab ärireegli metaandmed. Reegli käivitamisel interpreteerib mootor metaandmed ja käivitab ärireeglid.
- Reeglite mootoris salvestatakse metaandmed ja selle põhjal genereeritakse käivitatav kood.

Antud töö raames pakutavas lahenduses kasutatakse samuti lähenemist, kus kasutaja defineerib reegli käivitataval kujul teenuse sisendina. Chisholm [18], kes kirjutab kuidas luu ärireeglite haldussüsteemi, seda lahendust lähemalt ei käsitle, kuna kasutaja jaoks võib olla selline lahendus keeruline. Käivitatava koodi sisendina defineerimise korral peab kasutaja tundma vähemalt mingis ulatuses vastavat programmeerimiskeelt ja selle omadusi, et tarkvara andmetega juhtida. Kuna antud töö aluseks oleva lahenduse puhul eeldatakse, et API kasutajad on arenduskompetentsiga, siis võib pidada sellist lähenemist nende jaoks andmetega juhitava teenuse arendamisel sobivaks. Chisholm [18] käsitleb lähemalt ja näidetega ärireeglite metaandmete-põhist esitust.

2.2 „Minimeeri koodi, maksimeeri andmeid“

Ken Downs kirjeldab põhimõtet “minimeeri koodi, maksimeeri andmeid” oma samanimelises ajaveebi postituses (“*Minimize code, Maximize data*”) [11] kõige paremini hoitud saladusena tarkvaraarenduses. Tema arvates võib selle idee vähese populaarsuse põhjuseks olla “programmeerija mentaliteet” – tarkvaraarendajad on harjunud, et muudatusi saab sisse viia läbi koodi kirjutamise.

Downsi ajaveebi postituses esitatud probleem tuleb isiklikult kogetud juhtumist. Ühe tema kliendi tarkvara vajas äriloogika täiendust, kuid arendaja ja klient ei julgenud muudatusi programmikoodis teha, sest nad kartsid, et tekivad vead. Tegu on ajakirjade jaotussüsteemiga, mille ajakirjade koguseid reguleeriv loogika muutus ja mida olemasolev programmiloogika enam ei lahendanud.
Downs esitleb võimalust juhtida regulatsioonitarkvara andmetega kasutades otsustustabeleid. Ta kirjeldab ärireeglit, kus vähem kui 20% müügitulemusega ajakirja puhul tuleb vähendada ajakirja kogust järgmiseks jaotuseks kahe ühiku võrra, kuid mitte vähendada kogust alla kahe ühiku. Otsustustabelit ajakirja koguse vähendamiseks kirjeldab Tabel 1 [11].

Tabel 1: Otsustustabel ajakirjade koguse vähendamiseks.

THRESHOLD_PERCENT	DECREMENT	ABSOLUTE_MINIMUM
20	2	2

Kui toodet müüdi jaotusperioodil enam kui 80% ulatuses, siis suurendatakse järgmiseks perioodiks toote kogust kahe ühiku võrra. Edukate müügitulemuste jaoks on esitatud otsustustabel Tabel 2.

Tabel 2: Otsustustabel ajakirjade koguse suurendamiseks.

THRESHOLD_PERCENT	INCREMENT
80	2

Otsustustabelites sisalduvate reeglite alusel tehakse regulatsiooni käivitamise hetkel otsuseid. Regulatsiooni käigus itereeritakse üle ajakirjade massiivi. Iga ajakirja puhul kontrollitakse, kas müügitulemused vastavad esitatud reeglile. Kui kontrolli tulemus vastab reegli kirjeldusele, siis muudetakse vastavalt toodete hulka.

2.3 Andmetega juhitava tarkvara näiteid

Süsteemi käitumise andmetega juhtimist on käsitletud ka magistritöös, mille raames loodi PostgreSQL andmebaasis käivitatavate SQL lausete paindlikuks muutmiseks mõeldud tarkvara [21]. Nimetatud töö tulemuseks on eelkõige analüütikutele suunatud rakendus, mis võimaldab vastavalt muutunud nõuetele muuta SQL lausete koodi graafilise liidese kaudu. Töö dünaamiliste päringute koostamise graafiline liides võiks olla eeskujuks ja aluseks käesolevas töös pakutavale lahendusele võimaliku graafilise liidese loomisel.

Veel üheks andmetega juhitava tarkvara näiteks on veebirakenduste arenduskeskkonnad nagu Oracle APEX [22] ja analoogiliselt PostgreSQL jaoks mõeldud pgApex [23] [24]. Need veebirakenduste arendus- ja käituskeskkonnad on metaandmetega juhitavad kiirprogrammeerimise keskkonnad. Graafilise liidese abil loovad arendajad veebilehitseja vahendusel veebirakenduse. Rakenduse kirjeldus (lehtede struktuur, kasutajaliidese elementidega seotud käitumine, rakenduse väljanägemine, turvalisuse küsimused jms) salvestatakse andmebaasis. Kui loodud rakenduse lõppkasutaja pöördub rakenduse lehe poole, siis loeb tarkvara mootor andmebaasist selle lehe kirjelduse ja

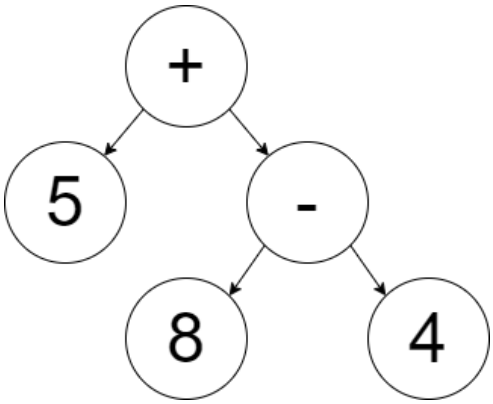
andmed, mida tuleb kasutajale selle lehe kaudu esitada. Mistahes muudatus rakenduses on lõppkasutajale nähtav niipea kui vastav muudatus on arenduskeskkonna kaudu andmebaasis salvestatud. Sellise tarkvara kasutamiseks on väga oluline hea graafiline kasutajaliides. „pgApex esimese versiooni andmebaasis oli 36 tabelit ja 176 veergu. Käesoleva lõputöö tulemusena loodi 16 tabelit, milles on kokku 119 veergu.“ [24] Rakenduse loomine „käsitsi“ oleks selle näite põhjal väga veaohtlik ning ebaotstarbekas, sest andmebaasi struktuur on liiga keeruline.

2.4 Vajadupõhine integratsioon

Üha rohkem organisatsioone võtab kasutusele SaaS lahendusel põhinevat tarkvara ja tekkinud on nõudlus vajaduspõhise integratsiooni (*integration on demand*) järele. See tähendab, et mittetehnilise taustaga klientidel on võimalus veebilehitseja-põhiste lahenduste kaudu kohandada ja paigaldada integratsioone SaaS arhitektuuril põhinevate teenustega [25]. Käesoleva töö peamiseks sihtgrupiks andmetega teenuste juhtijate mõistes on teenuseid kasutavad arenduskompetentsiga inimesed nagu näiteks tarkvaraarendajad, kes integreerivad arendatavasse süsteemi teenuse liidese. Antud töö raames valmiv lahendus ja disainimuster võiksid olla aluseks võimalikule edasiarendusele, kus graafilise liidese kaudu on võimalik lõppkasutajal klientsüsteemis teenuste käitumist ja äriloogika toimist suunavaid avaldisi kohandada.

2.5 C# avaldistepuu ja LINQ

Avaldistepuu C# keeles on koodi kirjeldav puusarnane andmestruktuur, kus iga sõlm on käivitatav avaldis [26]. Joonis 1 kujutab C# avaldistepuud *Add*(*Constant*(5), *Subtract*(*Constant*(8), *Constant*(4))), lihtsustatult avaldist 5 + (8 – 4). Avaldistepuu abil on võimalik C# programmeerimiskeeles defineerida dünaamilisi arvutuskäike esitavaid protsesse. Avaldiste hulka kuuluvad avaldiste (*Expression*) abstraktse baasklassi derivaadid, mis moodustavad avaldistepuu sõlmed [27].



Joonis 2: Avaldistepuu andmestruktuuri skemaatiline esitus.

Üheks tavalisemaks avaldiste näiteks on atribuudi või andmevälja dünaamiline pärimine sisendiks oleva sõne põhjal. Joonis 3 on näide mingi objekti (*entity*) välja nimega "name" väärtuse küsimisest avaldise abil.

```
var fieldName = "name";
Expression.Property(entity, fieldName);
```

Joonis 3: Objekti välja väärtuse avaldis.

Kuna välja nimi joonisel esitatud näites on dünaamiliselt muudetav ning objekti välja poole pöördutakse avaldiste kaudu, siis on võimalik programmi käivitamise hetkel "*fieldName*" väärtust muuta ning pöörduda seeläbi sama objekti mõne teise välja poole.

Mitut avaldist koos on võimalik esitada avaldistepuuna. Joonis 4 on esitatud *if-else* (siis-kui) plokk avaldistepuuna, kus muutuja "*booleanValue*" tõese väärtuse korral tagastatakse sõne "*Tõene*", väära väärtuse korral "*Väär*":

```
var booleanValue = true;
Expression.IfThenElse(
      Expression.Constant(booleanValue),
      Expression.Constant("Tõene"),
      Expression.Constant("Väär")
)
```

Joonis 4: Tingimusavaldise puu.

Reeglitepõhine filtreerimine programmi mälus olevate objektide kogumi põhjal on varasemalt lahendatud C# avaldistepuu struktuuri läbi. Selliste süsteemide põhimõtteks on massiivist väärtuste filtreerimine vastavalt avaldistes esitatud tingimustele [26] [28]. Antud töö mõistes on oluline võimaldada avaldiste kaudu lugeda ja muuta andmebaasis olevaid väärtuseid programmi operatsioonide käivitamise hetkel. Näidisprojektis

kasutatakse andmebaasipäringute ja avaldiste koostamiseks LINQ päringukomponendi meetodeid [29].

2.6 Lõppkasutaja arendamise printsiip

Tarkvara andmetega juhitavaks muutmine oleks esimene samm lõppkasutajapoolse arendamise printsiibi (*end-user development*) [30] [31] rakendamiseks. Selle kohaselt osalevad tarkvara lõppkasutajad ise aktiivselt enda kasutatava tarkvara loomises või kohandamises. Selliste rakenduste näideteks on erinevad prototüüpide loomise tarkvarad. Käesolevas töös väljapakutav lahendus ei ole mõeldud tarkvara lõppkasutajatele, vaid IT spetsialistidele. Need võivad olla tarkvarateenuse pakkuja teenistuses olevad arendajad, aga ka kliendi IT-töötajad, kelle ülesandeks on REST teenuste integreerimine mingisse arendatavasse süsteemi. Süsteemiarenduse ülesannete üleandmine tarkvaraettevõttelt teenuse tarbijatele oleks lõppkasutaja arenduse saavutamise esimene samm. Tõelise lõppkasutaja arenduse jaoks on vajalik luua vähese-koodiga või koodi loomise vajaduseta (*low-code/no-code development platform*) arenduskeskkonnad, millel on lihtne ja loogiline graafiline kasutajaliides [32]. See ei ole käesoleva töö eesmärk, kuid on kindlasti üks oluline töö jätkamise suund.

2.7 GraphQL

GraphQL on päringukeel API-dele, mis võimaldab päringu sisendis täpsustada vastuse struktuuri. GraphQL on Facebooki arendatud ning selle üheks olulisemaks põhimõtteks on asendada mitu “rumalat” API liidest ühe “targa” liidesega. GraphQL aluseks olev skeem kirjeldab andmestruktuuri ja andmete pärimise operatsioonid, olles dünaamiliseks vahelüliks kliendi ja andmete ligipääsukihi vahel [33] [34].

Võrdluses tavaliste REST teenustega on GraphQL ja kompaktsem, kuna võimaldab asendada mitu päringut ühe päringuga. Lihtsustatud kujul on GraphQL päringukeel küsimaks kindlaid skeemiga kirjeldatud objektide väljasid ilma vajaduseta pärida objektide kõiki andmeid. GraphQL dokumentatsiooni näitena on siinkohal toodud “kangelase” päring [35]. Süsteemis on registreeritud üks kangelane.

Joonis 5: GraphQL päring defineeritud väljadega ning API vastus [35].

Nagu on Joonis 5 näha, siis andmeallikast päritakse kangelase objekte ja nende puhul ainult nime andmevälja väärtust. Kui eeldada, et andmeallikas on kangelastel lisaks nimele kirjeldatud ka vanus ja sugu, siis GraphQL tagastab ainult päringus soovitud väljade väärtused. GraphQL suureks plussiks on võimekus asendada mitu REST päringut ühe pesastatud (*nested*) päringuna [35]. Joonis 6 kujutatud päring pärib kangelaste objektidega seotud sõprade nimekirju pesastatud päringuga.

Joonis 6: GraphQL pesastatud päring defineeritud väljadega ning API vastus [35].

Kui siin tuua võrdluseks klassikalisel kujul REST päringud, siis Joonis 7 esitab esimese päringu ja selle vastuse ning Joonis 8 esitab teise päringu ja selle vastuse. REST päringu näidete korral tagastab päring ka andmeid, mida GraphQL sama päringu korral tagastatavas andmestruktuuris ei defineeritud. Kui REST teenuste puhul oleks vajadus „kangelase“ päringu korral eristada kasutajatele loetavate atribuutide hulka (ühele kasutajale kuvada kogu kangelase info, teisele kuvada vaid kangelase nimi), tuleks selleks luua erinevad liidese lõppsõlmed.

```
GET: https://URL/hero
{
      "hero": {
            "name": "R2-D2",
            "age": 10,
            "gender": "male"
      }
}
```

Joonis 7: REST API "kangelase" päring ja selle vastus.

```
GET: https://URL/hero/friends

{
      "friends": [
            {
                  "name": "Luke Skywalker",
                  "age": 27,
                  "gender": "male"
            },
            ...
      ]
}
```

Joonis 8: REST API "kangelase" sõprade päring.

Näidetes on toodud välja võimalikud lisaväljad koos andmetega, mis on mõeldud vaid illustreerimaks REST API ja GraphQL erinevusi.

C#/.NET programmeerimiskeele jaoks on mitu erinevat raamistikku GraphQL intergeerimiseks. Nagu ülalpool kirjeldatud näidete alusel on ka nende raamistike alusel võimalik siduda GraphQL päring andmeallikaga ning vastavalt kirjeldatud skeemile teha väljade mõistes andmetega juhitavaid päringuid. Selliselt on näiteks EntityGraphQL raamistikul seotud andmeallikaga üks POST tüüpi kontroller, mis kuulab päringuid ning vastavalt GraphQL skeemile tagastab vastused. Joonis 9 olev koodilõik on EntityGraphQL näide kontrolleri struktuurist, mis võtab vastu API päringuid [36] [37].

```
[HttpPost]
public object Post([FromBody]QueryRequest query)
{
       try
       {
             var results = _schemaProvider.ExecuteQuery(
                                    query, _dbContext, null, null);
             return results;
       }
       catch (Exception) {
              return HttpStatusCode.InternalServerError;
       }
 }
```

Joonis 9: EntityGraphQL kontrolleri lõppsõlm.

Kuigi GraphQL on oma olemuselt päringukeel, on võimalik selle skeemis kirjeldada ka andmete uuendamise ja sisestamise operatsioonid. Andmetega juhitavuse mõistes on GraphQL üks paremaid olemasolevaid lahendusi, sest võimaldab päringu vastuse struktuuri sisendandmetega juhtida. Samas ei ole GraphQL päringute sisendi kaudu muudetav teenuse äriloogika probleemipüstituses kirjeldatud moel, sest sisendandmete kaudu pole võimalik defineerida äriloogilisi avaldisi.

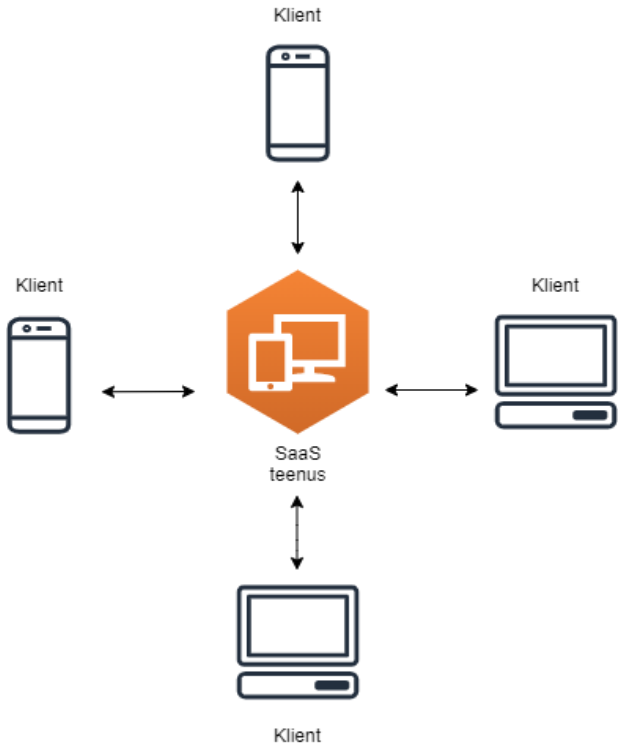
Andmetega juhitavus probleemipüstituses kirjeldatud viisil ja GraphQL-i võimalused oleksid teineteist täiendavad lahendused. Üks disain võimaldab andmete struktuuri dünaamiliselt päringutes defineerida, teine disain võimaldab andmete sisu ja nende aluseks olevat äriloogikat dünaamiliselt päringute sisendina defineerida.

2.8 Mitmevalduslik SaaS mudel ja andmete eraldatus

SaaS mudeli üheks kõige olulisemaks põhimõtteks on, et teenusepakkuja pakub programmi ja selle operatsioone mitmele kliendile. Mitmevalduslik arhitektuur SaaS ärimudelis võimaldab mitmel kasutajal/kliendil/süsteemil kasutada sama rakendust/liidest ning andmebaasi. Klientide jaoks tähendab SaaS teenuste integreerimine enda süsteemidesse vähenenud arendus- ja hoolduskulusid, kuna integreeritava teenuse arendus ja hooldus on teenusepakkuja kohustuseks. Olulised omadused mitmevalduslikus arhitektuuris on riistvararessursside jagamine, kõrge konfigureeritavus ning jagatud rakendus ja andmebaas [3].

Joonis 10 on kujutatud SaaS mudel, kus keskse teenusega suhtlevad erinevad klientsüsteemid. Need klientsüsteemid võivad üksteise suhtes olla konkureerivad

teenusepakkujad samale klientide segmendile enda tegevusvaldkonnas. SaaS teenuse vajadusest sobituda võimalikult suurele hulgale klientidele tuleneb ka mitmevalduslike süsteemide vajadus parema konfigureeritavuse, suurema andmete turvalisuse ja parema veahalduse järgi.



Joonis 10: SaaS mudel, mille keskmes on mitmevalduslik teenus.

Kuigi turvalisuse tase on ka üksikvalduslike rakenduste puhul väga oluline, on mitmevalduslike keskkondade puhul võimalik risk turvalisusega seotud probleemideks suurem. Üksikvalduslikes süsteemides on andmete varastamise oht küll olemas, kuid suhteliselt madal. Mitmevalduslikes süsteemides on andmevarguste või võltsimiste oht suurem, sest reeglina on konkureerivate ettevõtete andmed teenusepakkuja poolel ühes andmebaasis. Ründajale oleks see suurem saak kui üksiku ettevõtte andmed. Lahendusena andmete turvalisuse probleemile on mitmevalduslikes süsteemides võimalik kasutada kohandatud versioone mitmevalduslikust süsteemist.

- Jagatud rakendus, eraldi andmebaas – igale teenuse kliendile on andmete hoiustamiseks eraldiseisev andmebaas. Rakenduse ühises andmebaasis hoiustatakse vaid klientide autentimiseks vajalikke andmeid ning viidet kliendi andmebaasile.

- Jagatud rakendus, jagatud andmebaas, eraldi tabel – igale teenuse kliendile on loodud jagatud andmebaasis kõikidest tabelitest vastava kliendi ID-ga täiendatud nimetusega tabel.

Puhas mitmevalduslik mudel on selline, kus on jagatud rakendus, jagatud andmebaas ning ka jagatud tabelid/skeemid – andmete erinevus tuleb üldiselt lisatud kliendi ID andmeväljast [2] [3].

3 Regulatsiooniprogrammi analüüs ja arendus

Antud peatükk hõlmab töös käsitletava lähenemise *näidisena* valmiva regulatsioonitarkvara kavandamist ja arendust. Kavandamise käigus kirjeldatakse probleem ning pakutakse võimalik lahendus. Arenduse käigus realiseeritakse C# keeles REST API liides koos äriloogika ja andmebaasi struktuuriga. Antud peatükis ei kohandata veel rakendust andmetega juhitavaks.

3.1 Regulatsioonitarkvara kirjeldus

Ken Downs kirjeldab oma ajaveebi postituses "*Minimize code, Maximize data*" [11] ajakirjade levitamisega tegeleva kliendi süsteemi läbi regulatsiooniprotsessi. Ajakirjade jaotusvõrgus viiakse edasimüüjatele laiali ajakirjad nende tellitud koguses, mis ideaalsel juhul oleks alati võrdsed müüdava kogusega. Reaalsuses aga tagastatakse levitajale 50-80% ajakirjadest, mida edasimüüja müüa ei suutnud. Sellisest ajakirjade ülejagamisest tuleneb ka vajadus reguleerida enne järgmist jaotusperioodi kogused selliselt, et need oleks ligilähedasemad reaalselt müüdavale kogusele.

Regulatsiooniprogrammi põhimõte töötab selliselt, et pärast edasimüüja poolset ajakirjade tagastust käivitatakse regulatsioon, mis võtab aluseks jaotatud koguse ja tagastatud koguse. Vastavalt kui tagastatud kogus on liiga suur osahulk esialgsest kogusest, siis jaotatakse järgmisel korral antud tellijale vähem ajakirju. Kui aga edasimüüja müüb kõik või peaaegu kõik ajakirjad, siis järgmisel korral seda kogust suurendatakse.

Ken Downs kirjeldab täpsustatud ärilist vajadust selliselt, et kui ajakirjade edasimüüja müüb väiksema koguse kui 20% ajakirjadest, siis vähendatakse järgmiseks korraks kogust kahe ühiku võrra, kuid jaotatud ajakirjade kogus ei tohi olla vähem kui kaks ühikut. Kui edasimüüja müüb rohkem kui 80% ajakirjadest, siis tuleb järgmisel korral suurendada kogust kahe ühiku võrra [11].

Antud töö raames laiendatakse regulatsioonitarkvara arhitektuuri mitmevalduslikule teenuste mudelile. Alusprobleemi mõistes tähendab see, et kui kirjelduses oli programmi

omanikuks ajakirjade jaotusega tegelev üks organisatsioon, siis siin pakutava tarkvara omanikuks on teenusepakkuja. Teenusepakkuja kliendid, süsteemi suhtes valdajad, on ajakirjade jaotusega tegelevad organisatsioonid.

Loodava süsteemi puhul pakub teenusepakkuja jaotustarkvara teenustena koos andmete hoiustamisega ning teenuse klientidel on võimalik integreerida teenuse välised REST liidesed vastavalt enda süsteemidesse. Sellise süsteemi puhul on andmetega juhtimise vajadus suurenenud ja enam õigustatud, kuna erinevatel klientsüsteemidel võivad olla erinevad ärilised protsessid ja vajadused. Sellisteks muutuvate nõuetega vajadusteks võivad olla näiteks toote hindade, koguhindade või koguste arvutusprotsessid. Lisaks ei ole kirjeldatav teenus mitte kuidagi piiratud ainult ajakirjade jaotussüsteemiga. Jaotussüsteemi teenust võib kasutada mistahes toodete info haldamiseks ning jaotuste ja regulatsioonide teostamiseks.

3.2 Skoobi kirjeldus ja kasutusjuhud

Antud jaotises kirjeldatakse arendatava rakenduse skoop ja peamised kasutusjuhud. Kasutusjuhud on kirjeldatud regulatsiooniprotsessi jaoks oluliste operatsioonide ulatuses – laiemat funktsionaalsust jaotusprogrammi üldiste omaduste osas antud töös ei kirjeldata ega arendata. Jaotuse ja tagastusega seotud funktsionaalsuste aluseks olevad tellimused ja edasimüüjad, nende loomine ja haldus, on antud näite põhjal skoobist väljas – vajalikud andmed on lisatud andmebaasi loomise hetkel INSERT INTO lausetega, mis genereeritakse Entity Framework [38] raamistiku poolt.

3.2.1 Süsteemi eesmärgid

- Edasimüüja poolt tellitud ajakirjade jaotusplaanide loomise, muutmise ja kustutamise võimaldamine.
- Edasimüüja tagastatud müümata ajakirjade tagastuste loomise, muutmise ja kustutamise võimaldamine.
- Regulatsioonide rakendamine edasimüüja tellitud ajakirjade kogustele kliendi poolt tagastatud ajakirjade hulga põhjal.

3.2.2 Põhiobjektid

- Pood (*Store*) – Edasimüüja (klient), mis on perioodiliselt tellinud ajakirjad edasimüümiseks.

- Jaotus (*Distribution*) – Poele tellimuse põhjal loodud jaotuskava poe tellitud toodetega ja nende kogustega.
- Tagastus (*Return*) – Poe tagastatud tooted.

3.2.3 Põhiprotsessid

- Jaotuse loomine, lugemine, muutmine, kustutamine.
- Tagastuse loomine, lugemine, muutmine, kustutamine.
- Regulatsiooni käivitamine.

3.2.4 Põhilised sündmused

- Ajakirjade levitaja loob uue jaotuse ühele edasimüüjale tellitud toodete ja koguste alusel.
- Ajakirjade edasimüüja ei müünud kõiki levitajalt saadud ajakirju müügiperioodil ning tagastab need levitajale.
- Ajakirjade levitaja käivitab regulatsiooni, et arvutada ümber edasisteks perioodideks edasimüüjale jaotatavad kogused.

3.2.5 Tegutsejad

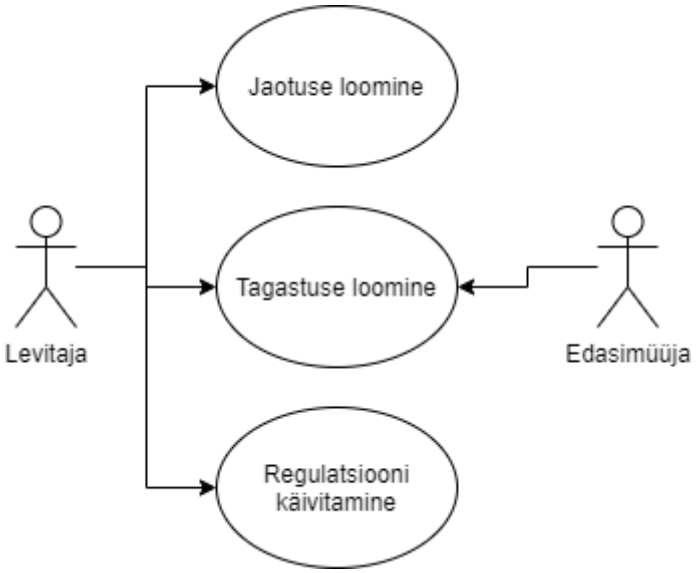
Antud töö käigus arendatava programmi nõuete kirjeldamiseks on olulised järgmised tegutsejad.

- Teenusepakkuja – teenuse arendaja ja hooldaja, tarkvara omanik.
- Levitaja – jaotusvõrgu haldaja, kes pakub ajakirjade hulgimüügi ja levitamise teenust edasimüüjatele. Teenusepakkuja mõistes klient, kes haldab oma jaotusvõrku ja sellega seonduvat infot teenusepakkuja ressurssidega.
- Edasimüüja – jaotusvõrgu mõistes klient, kes on tellinud endale perioodiliti mingi hulga ajakirju edasimüümiseks.

Levitaja on antud süsteemi ja teenusepakkuja mõistes klient, kes on teenusepakkuja programmi teenused integreerinud enda süsteemi, olles seeläbi ise integreeritud teenuste vahendaja enda klientide (edasimüüjate) jaoks, kes kasutavad läbi klientrakenduse teenuse tellimuste loomise ning tagastusega seotud funktsionaalsust. Jaotises 2.8 kirjeldatud mitmevalduslikku süsteemi joonisel (Joonis 10) on klientidena käsitletavad siin punktis kirjeldatud levitajate süsteemid ja programmid, mis tarbivad teenusepakkuja API-t.

3.2.6 Kasutusjuhtude mudel

Selles jaotises tuuakse välja kolm põhilist kasutusjuhtu, milledeks on ajakirjade jaotuse loomine, jaotuse alusel tagastuse loomine (POST operatsioonid) ning tagastuse järgselt regulatsiooni käivitamine. Kasutusjuhtudena on rakenduses lisaks realiseeritud ka jaotuse ja tagastuse GET, UPDATE ja DELETE operatsioonid, kuid kasutusjuhtude mudelil ning analüüsis neid ei kajastata.



Joonis 11: Teenuse põhiliste kasutusjuhtude diagramm.

Joonis 11 on kujutatud arendatava teenuse kasutusjuhtude diagramm koos kolme põhilise kasutusjuhuga kahe teenust tarbiva tegutseja jaoks. Jaotuse loomine ja tagastuse loomine on antud kontekstis väiksema tähtsusega kasutusjuhud ja pigem toetavad probleemipüstituses esitatud eesmärkide suhtes. Peamine kasutusjuht, mille põhjal peatükis 4 ka teostatakse teenuse andmetega juhitavaks muutmine, on regulatsiooni käivitamise protsess. Tabel 3 kuni Tabel 5 esitavad nende kasutusjuhtude laiendatud formaadis kirjelduse.

Tabel 3: Kasutusjuht UC-1 Jaotuse loomine.

ID	UC-1
Nimi	Jaotuse loomine
Kirjeldus	Kui levitaja kliendiks olev edasimüüja (pood) on tellinud endale kokkulepitud perioodis hulga ajakirjasid müümiseks, siis enne

	müügiperioodi algust ja ajakirjade transporti edasimüüja juurde loob levitaja süsteemis jaotuse. Jaotuse käigus pannakse kokku tellitud ajakirjade ja koguste põhjal nimekiri, milliseid tooteid ja mis koguses edasimüüjale toimetatakse
Tegutsejad	Levitaja
Eeltingimused	• Edasimüüja on registreeritud süsteemis kliendiks (*Store*) • Edasimüüja on tellinud ajakirjad (*StoreProducts*)
Põhivoog	1. Levitaja käivitab jaotuse operatsiooni edastades API-le edasimüüja poe ID 2. Süsteem kontrollib, kas vastava ID-ga pood on olemas 3. Süsteem kontrollib, kas pood on tellinud levitajalt ajakirju 4. Süsteem genereerib jaotuse poe tellitud ajakirjade ja kogustega
Laiendused	2a. Süsteem ei leia vastava ID-ga poodi ning tagastab veateate 3a. Süsteem tuvastab, et pood pole levitajalt tellinud ajakirjasid ning tagastab veateate
Järeltingimused	Jaotus koos toodete ja kogustega on lisatud andmebaasi ning tagastatakse API vastuses

Tabel 4: Kasutusjuht UC-2 tagastuse loomine.

ID	UC-2
Nimi	Tagastuse loomine
Kirjeldus	Kui edasimüüjal (poel) ei õnnestunud müügiperioodi jooksul mingit hulka jaotuse käigus saadud ajakirjadest müüa, siis tagastatakse need levitajale. Tagastuse käigus seotakse tagastatud tooted jaotusega.
Tegutsejad	Edasimüüja

Eeltingimused	• Jaotus (*Distribution*) on loodud kasutusjuhus UC-1 • Edasimüüjal on jaotuse käigus saadud toodetest (*DistributionProducts*) vaja midagi tagastada (*ReturnProducts*)
Põhivoog	1. Edasimüüja käivitab tagastuse operatsiooni edastades API-le poe ID, jaotuse ID ja massiivi tagastatavatest toodetest 2. Süsteem kontrollib, kas vastava ID-ga pood on olemas 3. Süsteem kontrollib, kas vastava ID-ga jaotus on olemas 4. Süsteem kontrollib, kas jaotusel on olemas tagastatud ajakirjad 5. Süsteem genereerib tagastuse poe tagastatud ajakirjade ja kogustega, sidudes tagastuse jaotusega
Laiendused	2a. Süsteem ei leia vastava ID-ga poodi ning tagastab veateate 3a. Süsteem ei leia vastava ID-ga jaotust ning tagastab veateate 4a. Süsteem tuvastab, et poel puuduvad tellitud tooted ning tagastab veateate
Järeltingimused	Tagastus koos toodete ja kogustega on lisatud andmebaasi ning tagastatakse API vastuses

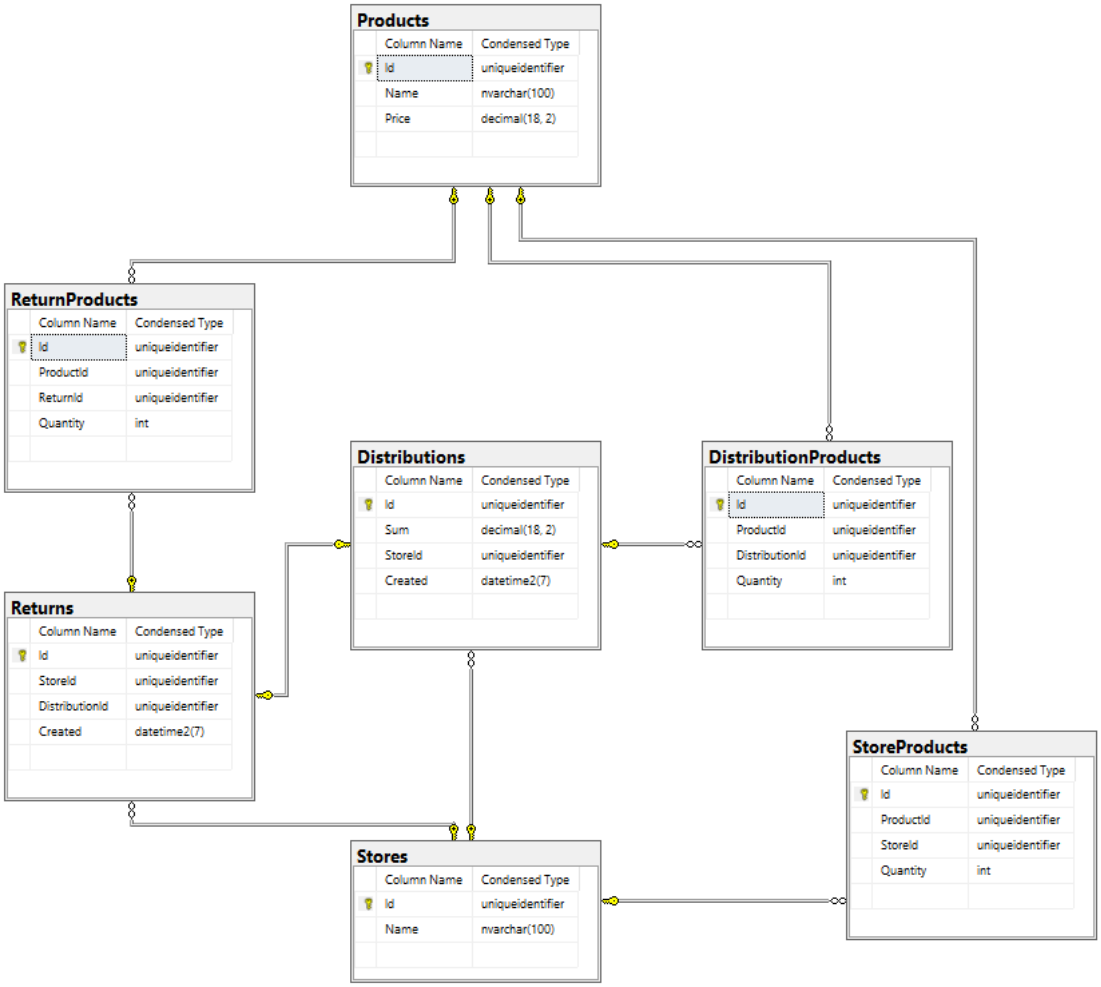
Tabel 5: Kasutusjuht UC-3 Regulatsiooni käivitamine.

ID	UC-3
Nimi	Regulatsiooni käivitamine
Kirjeldus	Levitaja käivitab edasimüüja (poe) tagastatud toodete ja loodud tagastuse põhjal regulatsiooni, et ümber arvutada poe tellitud ajakirjade kogused järgmiseks müügiperioodiks (*StoreProduct*). Regulatsioon käivitatakse viimase jaotuse ja sellele vastava tagastuse põhjal enne uue jaotuse loomist. Olenevalt klientsüsteemi ja jaotusperioodide eripärast võib regulatsiooni käivitada varasemate jaotuste põhjal, kui jaotusele vastavate toodete

	tagastamise hetkeks on vahepeal loodud uusi jaotusi samale edasimüüjale. Regulatsioon käivitatakse seega viimase jaotuse alusel, millele on loodud tagastus.
Tegutsejad	Levitaja
Eeltingimused	• Edasimüüja (pood) on registreeritud süsteemis kliendiks (*Store*) • Jaotus (*Distribution*) on loodud kasutusjuhus UC-1 • Tagastus (*Return*) on loodud kasutusjuhus UC-2
Põhivoog	1. Levitaja käivitab regulatsiooni operatsiooni edastades API-le kasutusjuhus UC-1 loodud jaotuse ID, millele vastavalt on loodud kasutusjuhus UC-2 tagastus. 2. Süsteem kontrollib, kas vastava ID-ga jaotus on olemas 3. Süsteem itereerib üle ajakirjade massiivi, mille pood on tellinud. Kui jaotusele vastavas tagastuses leidub vastav ajakiri, arvutatakse tagastatud ajakirjade osa jaotatud ajakirjadest: a. Kui pood on müünud alla 20% ajakirjadest, siis vähendatakse ajakirja kogust järgmiseks jaotuseks kahe võrra b. Kui pood on müünud üle 80% ajakirjadest, siis suurendatakse ajakirja kogust järgmiseks jaotuseks kahe võrra
Laiendused	2a. Süsteem ei leia vastava IDga jaotust ning tagastab veateate 3c. Kui tagastuses ei leidu mõnda ajakirja, mis leidus vastavas jaotuses, suurendatakse selle ajakirja kogust järgmiseks jaotuseks kahe võrra, kuna müügiprotsent on 100%
Järeltingimused	Poe tellitud ajakirjade kogused (*StoreProduct*) on ümber arvutatud vastavalt müügikoguse reeglitele ja salvestatud andmebaasis. API tagastab tulemusena poe tellitud ajakirjade massiivi koos ümberarvutatud kogustega

3.2.7 Andmemudel

Antud jaotises vaadeldakse näidisprojekti jaoks olulist osa jaotussüsteemi disaini andmemudelist. Andmemudel on lihtsustatud kujul ning kirjeldab vaid neid veerge, mis näidisprojekti vaatest on olulised.



Joonis 12: Regulatsioonisüsteemi lihtsustatud andmemudel.

Joonis 12 on regulatsiooniprogrammi osasüsteemi SQL-andmebaasi disaini kirjeldav diagramm. Tabel 6 kirjeldab tabelites olevate andmete tähendust (mis andmed seal on ja milleks neid vaja läheb).

Tabel 6: Tabelites olevate andmete kirjeldus.

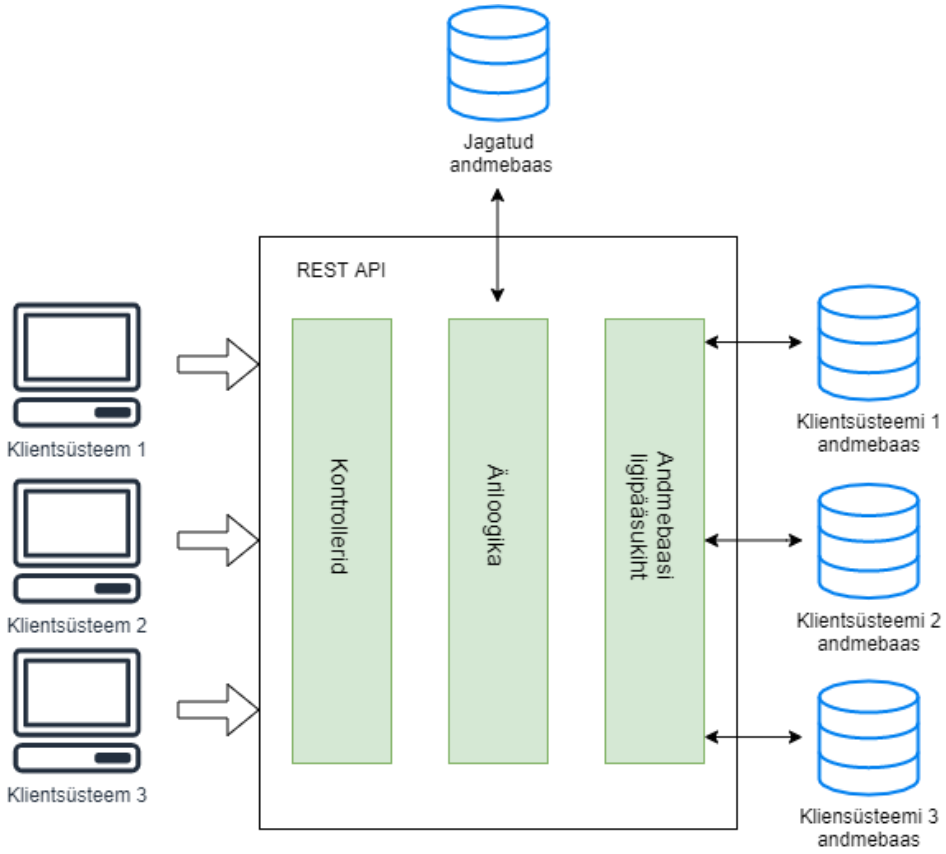
Tabeli nimi	Kirjeldus
Distributions	Jaotused, millega fikseeritakse poele jaotatavad tooted ja nende hulk.
DistributionProducts	Seostabel jaotuse ja toote vahel, milles defineeritakse jaotatavate toodete kogus. Realiseerib mitu-mitmele seost jaotuse ja toote vahel. Üks ja sama toode võib olla mitmes jaotuses ning ühes jaotuses võib olla mitu toodet. Samas jaotuses ei tohi toodet olla mitmekordselt. Seega tabelis on unikaalsuse kitsendus UNIQUE (Productid, Distributionid).
Products	Levitaja defineeritud tooted, mida on võimalik poel tellida.
Returns	Tagastused, millega fikseeritakse vastava jaotuse järgselt tagastatud tooted ja nende hulk.
ReturnProducts	Seostabel tagastuse ja toote vahel, milles defineeritakse tagastatud toodete kogus. Realiseerib mitu-mitmele seost tagastuse ja toote vahel. Üks ja sama toode võib olla mitmes tagastuses ning ühes tagastuses võib olla mitu toodet. Samas tagastuses ei tohi toodet olla mitmekordselt. Seega tabelis on unikaalsuse kitsendus UNIQUE (Productid, Returnid).
Stores	Pood, süsteemi mõistes ajakirjade tellija ja levitaja jaoks klient.
StoreProducts	Seostabel poe ja toote vahel, milles defineeritakse poe tellitud toodete kogus. Realiseerib mitu-mitmele seost poe ja toote vahel. Üks ja sama toode võib olla tellitud mitme poe poolt ning üks pood võib tellida mitu toodet. Sama poe poe tellimuses ei tohi toodet olla mitmekordselt. Seega tabelis on unikaalsuse kitsendus UNIQUE (Productid, Storeid).

Poodide (*Stores*) ja tellitud ajakirjade (*StoreProducts*) loomise ning haldamise funktsionaalsust antud näidisprogrammi raames ei arendatud. Programmi töötamiseks

vajalikud, kuid süsteemi kaudu mitte loodavad lähteandmed on nendesse tabelitesse lisatud andmebaasi loomisel.

3.3 Regulatsiooniprogrammi tehniline lahendus

Regulatsiooniprogrammi teenuste süsteemi [39] arhitektuuri iseloomustab Joonis 13.



Joonis 13: Süsteemi arhitektuuriline mudel.

Jooniselt on näha mitmevalduslikule süsteemile omast ülesehitust. Mitmevaldusliku süsteemi klientide suhtlust vahendab süsteemi äriloogikale kontrollerite kiht. Äriloogika kihi ülesandeks on esmalt klientsüsteemile vastava andmebaasiga ühenduse loomine. Selleks on rakendusel jagatud andmebaas, kus hoiustatakse klientsüsteemide andmeid. Kui kliendi ID-le vastava andmebaasiga on ühendus loodud, siis teostatakse kontrolleri päringule vastav operatsioon ning tehakse päring või salvestatakse andmed andmebaasi. Antud näites on igal kliendil oma andmebaas.

3.3.1 Prototüüp

Näidisprojekti esimeses iteratsioonis arendatakse antud töö näitel regulatsiooni tarkvara nii, et see vastab esialgsetele analüüsis püstitatud kasutusjuhtudele, kuid ei ole veel andmetega juhitav. Andmetega juhitavuse omadused lisatakse olemasolevale süsteemile peatükis 4. Prototüübi arendamiseks on kasutatud C# programmeerimiskeelel põhineva .NET Core raamistiku versiooni 3.1. Andmemudel on koostatud programmikoodis ning andmebaasi migratsioonid ja andmebaas on genereeritud programmikoodi põhjal (*code-first*). Objekt-relatsioonvastenduse (ORM – *object-relational mapping*) jaoks on kasutatud Entity Framework Core raamistikku (versioon 3.1.1), mille abil luuakse andmebaasimigratsioonid ja käivitatakse need ühendatud andmebaasis. Andmebaasisüsteemina kasutatakse töös MSSQL Serverit (versioon 2016). API dokumentatsiooni liides luuakse kasutades Swashbuckle.AspNetCore raamistikku (versioon 5.0.0). Arenduskeskkonnana kasutati Microsoft Visual Studio Community 2019 (versioon 16.5.4).

Äriprotsessidest kirjeldatakse antud peatükis üksikasjalikult regulatsiooni käivitamise protsessi realisatsiooni, mis on antud töö mõistes kesksek protsessiks. Regulatsiooniprotsessi põhjal realiseeritakse peatükis 4 andmetega juhitavaks muudetud protsess.

3.3.1.1 Kontrollerid

Vastavalt kirjeldatud kasutusjuhtudele saab API kontrollerid jagada kolmeks.

1. DistributionController – kontrollerikiht jaotusega seotud operatsioonide käivitamiseks (Joonis 14). Näidisprojekti põhjal on siin jaotuse loomise, lugemise, muutmise ja kustutamisega seotud API lõpp-punktid. Kasutusjuhtu kasutades (UC-1) on kirjeldatud sellele kontrolleri POST meetodi – jaotuse loomine – kasutamine.

Joonis 14: Jaotuse kontrolleri meetodite Swagger dokumentatsioon.

2. ReturnController – kontrollerikiht tagastustega seotud operatsioonide käivitamiseks (Joonis 15). Näidisprojekti põhjal on siin tagastuse loomise, lugemise, muutmise ja kustutamisega seotud API lõpp-punktid. Kasutusjuhtu kasutades (UC-2) on kirjeldatud sellele kontrolleri POST meetodi – tagastuse loomine – kasutamine.

Joonis 15: Tagastuse kontrolleri meetodite Swagger dokumentatsioon.

3. RegulationController – kontrollerikiht regulatsiooniprotsessi käivitamiseks (Joonis 16). Näidisprojekti vaadeldes reguleeritakse selle kontrolleri kaudu poe tellitud toodete koguseid järgmiseks jaotuseks. Kasutusjuhtu kasutades (UC-3) on kirjeldatud sellele kontrolleri POST meetodi – regulatsiooni käivitamine – kasutamine.

Joonis 16: Regulatsiooni kontrolleri meetodite Swagger dokumentatsioon.

3.3.1.2 Äriloogika kiht

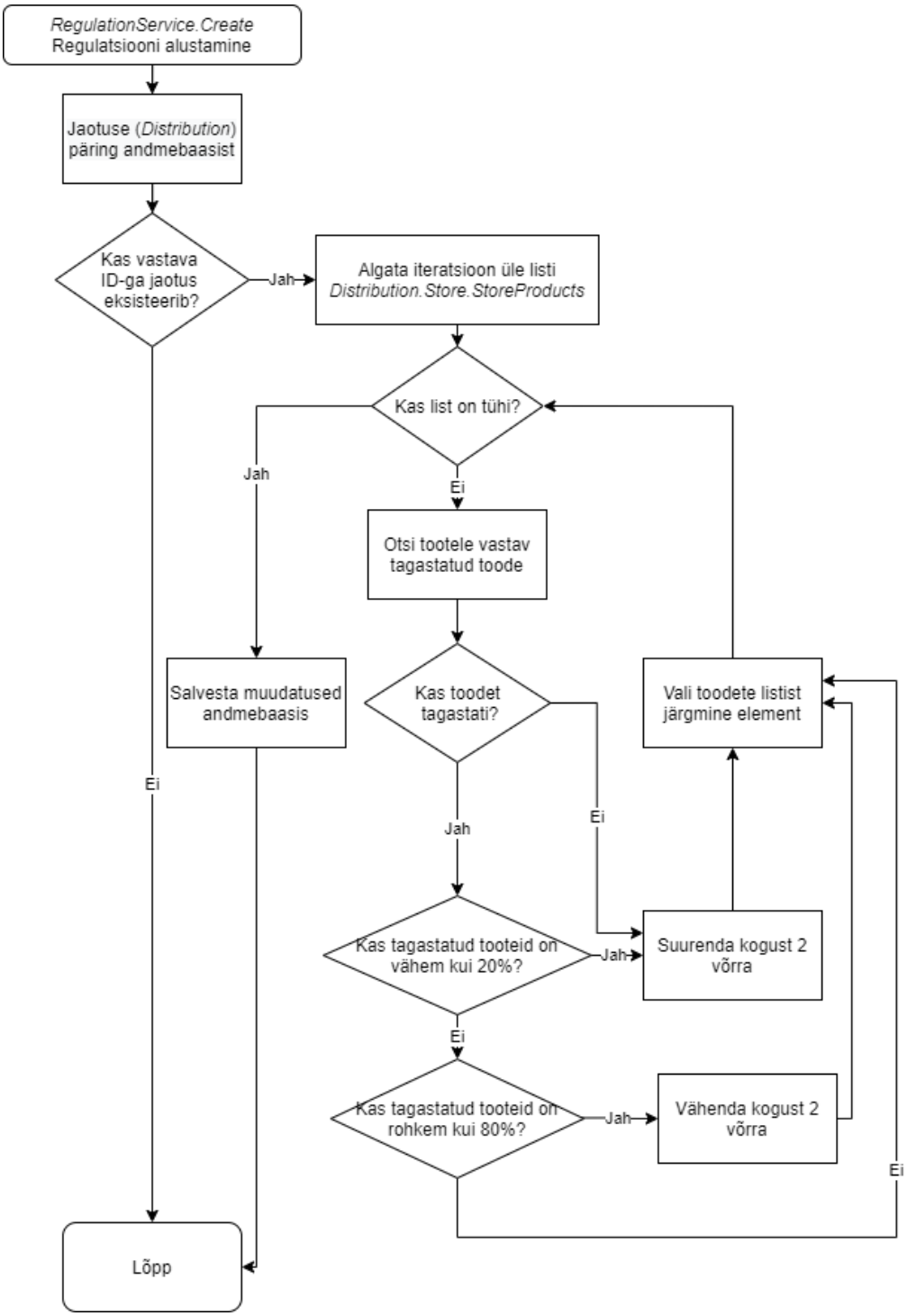
Äriloogika kiht on sarnaselt kontrolleritele jaotatav kolmeks teenuseks: DistributionService, ReturnService ning RegulationService. Iga teenuse ülesanne on võtta vastu vastava kontrolleri päringud ning käivitada ettenähtud äriprotsessid ja andmebaasi ligipääsu meetodid.

3.3.1.3 Andmebaas ja ORM

Näidisprojekti andmebaas on MSSQL andmebaasisüsteemis (MSSQL Server 2016) realiseeritud andmebaas ning andmebaasi genereerimiseks, migratsioonide loomiseks ja halduseks kasutatakse Entity Framework Core raamistikku (versioon 3.1.1). Nimetatud raamistiku puhul on oluliseks heaks omaduseks LINQ päringute tugi, mida kasutatakse näidisprojektis andmebaasipäringute tegemiseks [38].

3.3.1.4 Regulatsioon

Regulatsiooniprotsessi äriloogika operatsiooni ja töövoogu kirjeldab Joonis 17 vooskeem (*flow-diagram*). Käivitav sündmus ja lõppseisund esitatakse ümarate servadega ristkülikuna, toimingud teravate servadega ristkülikuna ning otsustuspunkt rombina.



Joonis 17: UC-3 kasutusjuhu vooskeem.

Antud töö mõistes on jaotuse ja tagastuse API liidesed toetavaks mehhanismiks põhilisele operatsioonile, milleks on regulatsioon. Seetõttu ei ole töös lahti kirjutatud täpsemalt esimese kahe nimetatud süsteemi osa funktsionaalsus. Joonis 17 on kirjeldatud peamise ärilise funktsionaalsuse vooskeem. Regulatsiooniprotsess käivitatakse levitaja poolse tegevusena ning selle aluseks on konkreetne jaotus. Kui levitaja jaoks on lõppenud

tagastuste vastuvõtmise periood ning tulemas on uue jaotuse periood, siis käivitatakse süsteemis Joonis 18 esitatud regulatsiooni päring.

```
POST: https://localhost:44353/regulation/distribution/{distribution_id}
```

Joonis 18: Regulatsiooni päring.

Joonis 19 esitatud koodilõik on regulatsiooni realisatsioon C# keeles kasutades LINQ päringukomponenti. Koodi kirjeldamiseks on ridadele lisatud kommentaarid.

```
public Store Create(Guid distributionId)
{
// Andmebaasist päritakse ID-le vastava jaotuse andmed
      var distribution = _regulationDbContext.Distributions
                .FirstOrDefault(d => d.Id == distributionId);

// Kui ID-le vastavat jaotust pole, siis programm peatatakse
      if (distribution == null) return null;

      var store = distribution.Store;

// Iteratsioon üle jaotuse olemiga seotud poe tellitud toodete
massiivi
      store.StoreProducts.ForEach(sp =>
      {
// Otsi tootele vastavat tagastatud toodet
            var returnProduct =
distribution.Return.ReturnProducts.FirstOrDefault(rp => rp.ProductId
== sp.ProductId);
// Kui otsitavat toodet tagastatud toodete hulgast ei leitud, siis
suurendatakse selle kogust järgmiseks jaotuseks kahe võrra
            if (returnProduct == null) {
                  sp.Quantity += 2;
                  return;
            }
// Kontroll, kas tagastatud toodet on enam kui 80% esialgsest
kogusest ja kas vähendamise korral on kogus enam kui 2 ühikut
            var decrement =
(returnProduct.Quantity / sp.Quantity) * 100 > 80
&& (sp.Quantity - 2) > 2;

// Kontroll, kas tagastatud toodet on vähem kui 20% esialgsest
kogusest
            var increment =
(returnProduct.Quantity / sp.Quantity) * 100 < 20;
// Koguse väärtustamine tellitud tootel: Kui kogust vähendatakse,
siis lahutatakse olemasolevast kogusest 2; kui kogust suurendatakse,
siis liidetakse olemasolevale kogusele 2; muul juhul jääb väärtus
samaks
            sp.Quantity = decrement ?
sp.Quantity -= 2 : increment ?
sp.Quantity += 2 : sp.Quantity;
        }
      );
// Andmete muudatused uuendatakse andmebaasi kontekstis
      store = _regulationDbContext.Update(store).Entity;

// Andmebaasi muudatused salvestatakse
      _regulationDbContext.SaveChanges();
      return store;
}
```

Joonis 19: Regulaatsiooniloogika koodinäidis.

4 Regulatsioonitarkvara andmetega juhitavus

Antud peatükis muudetakse peatükis 3 kirjeldatud analüüsi põhjal loodud tehnilist lahendust andmetega juhitavaks. Peatükis kirjeldatakse pakutav lahendus samm-sammult tuginedes peatükis 3 kirjeldatud regulatsiooniprotsessile, kus arvutatakse ümber tellitud toodete hulk. Peatüki lõpus hinnatakse pakutud lahendust ühe muutunud nõude kaudu, võrreldes originaalfunktsionaalsuse ja sisendi kaudu muudetud funktsionaalsuse kasutamise tulemusi.

Regulatsiooniprogramm, eesotsas regulatsiooni käivitamise protsessiga, on antud töö kontekstis süsteemi muutuv osa. Järgnev nimekiri kirjeldab *näitena* väikest hulka võimalikke muudatusi, mis võivad tarkvara elutsükli käigus ette tulla, kuid millele vastavat funktsionaalsust ei ole esialgsetes nõuetes ette nähtud.

- Toodete koguse ümberarvutus – Kui toodet tagastati enam kui kümme ühikut, siis vähendada toote kogust viie ühiku võrra.
- Toodete hinna ümberarvutus – Kui tagastatud toodet jaotuses ei ole, siis vastava toote hinda vähendatakse järgmiseks jaotuseks 2,5%.
- Jaotuse hinna ümberarvutus – kui koguhind ületab mingi summa piiri, siis tuleb rakendada koguhinnale soodustust 10%.

Käesolevas peatükis ei kirjeldata andmetega juhitavalt juba varasemalt realiseeritud funktsionaalsust – st kasutusjuhus UC-3 esitatud nõue toodete koguse ümberarvutamiseks on siin kontekstis põhifunktsionaalsuseks.

4.1 Muutunud ärinõue

Regulatsioonitarkvara uus versioon tõukub muutunud ärinõudest („toote koguse ümberarvutus“), mille kohaselt UC-3 (vt Tabel 5) kirjeldatud jaotuse reeglid enam ei kehti. Selle asemel on vaja võtta kasutusele uued reeglid – kui toodet tagastati enam kui kümme ühikut, siis vähendada toote kogust viie ühiku võrra. Peatükis 3 esitatud lahenduse korral tuleks teha muudatus programmikoodis (Joonis 19). Eesmärgiks on muuta tarkvara selliseks, et edaspidise sarnase muudatuse jaoks ei oleks vaja programmikoodi muuta. Jaotis 4.2 kirjeldab selle eesmärgi saavutamise võimalikke viise.

Lisandunud ärinõuet „toodete koguse ümberarvutus“ kirjeldab kasutusjuht UC-4 (Tabel 7).

Tabel 7: Kasutusjuht UC-4 Toodete koguse ümberarvutus.

ID	UC-4
Nimi	Toodete koguse ümberarvutus
Kirjeldus	Levitaja käivitab edasimüüja (poe) tagastatud toodete ja loodud tagastuse põhjal regulatsiooni, et ümber arvutada poe tellitud ajakirjade kogused järgmiseks müügiperioodiks. Regulatsioon käivitatakse viimase jaotuse ja sellele vastava tagastuse põhjal enne uue jaotuse loomist. Olenevalt klientsüsteemi ja jaotusperioodide eripärast võib regulatsiooni käivitada varasemate jaotuste põhjal, kui jaotusele vastavate toodete tagastamise hetkeks on vahepeal loodud uusi jaotusi samale edasimüüjale. Regulatsioon käivitatakse seega viimase jaotuse alusel, millele on loodud tagastus.
Tegutsejad	Levitaja
Eeltingimused	• Edasimüüja (pood) on registreeritud süsteemis kliendiks (*Store*) • Jaotus (*Distribution*) on loodud kasutusjuhus UC-1 • Tagastus (*Return*) on loodud kasutusjuhus UC-2
Põhivoog	1. Levitaja käivitab regulatsiooni operatsiooni edastades API-le kasutusjuhus UC-1 loodud jaotuse ID, millele vastavalt on loodud kasutusjuhus UC-2 tagastus. 2. Süsteem kontrollib, kas vastava ID-ga jaotus on olemas 3. Süsteem itereerib üle ajakirjade massiivi, mille pood on tellinud. Kui jaotusele vastavas tagastuses leidub vastav ajakiri, siis arvutatakse tagastatud ajakirjade osa jaotatud ajakirjadest: a. Kui pood tagastas jaotuse järel toodet enam kui kümme ühikut, vähendada järgmiseks jaotuseks kogust 5 võrra.

	b. Kui pood ei tagastanud toodet või tagastatud kogus on vähem kui 10 ühikut, jääb toodete kogus samaks
Laiendused	2a. Süsteem ei leia vastava IDga jaotust ning tagastab veateate
Järeltingimused	Poe tellitud ajakirjade kogused (*StoreProduct*) on ümber arvutatud vastavalt müügikoguse reeglitele ja salvestatud andmebaasis. API tagastab tulemusena poe tellitud ajakirjade massiivi koos ümberarvutatud kogustega

4.2 Regulatsiooniprotsessi andmetega juhtimise lahendusi

Selles peatükis vaadeldakse kahte regulatsiooniprotsessi andmetega juhtimise võimalust: Downsi [11] välja pakutud reeglitabelite lahendust ja käesoleva töö autori välja pakutud avaldistepuu lahendust.

4.2.1 Reeglitabel

Downsi [11] poolt pakutud lahenduse kohandamine rahuldamaks uut kirjeldatud nõuet eeldab uue otsustustabeli kasutuselevõtmist. Otsustustabeli loomise käigus on vaja defineerida lävendkogus, mida ületades vähendatakse järgmiseks jaotuseks toodet viie ühiku võrra. Tabelina on see esitatav nii nagu tehakse Tabel 8.

Tabel 8: Otsustustabel ajakirjade koguse vähendamiseks.

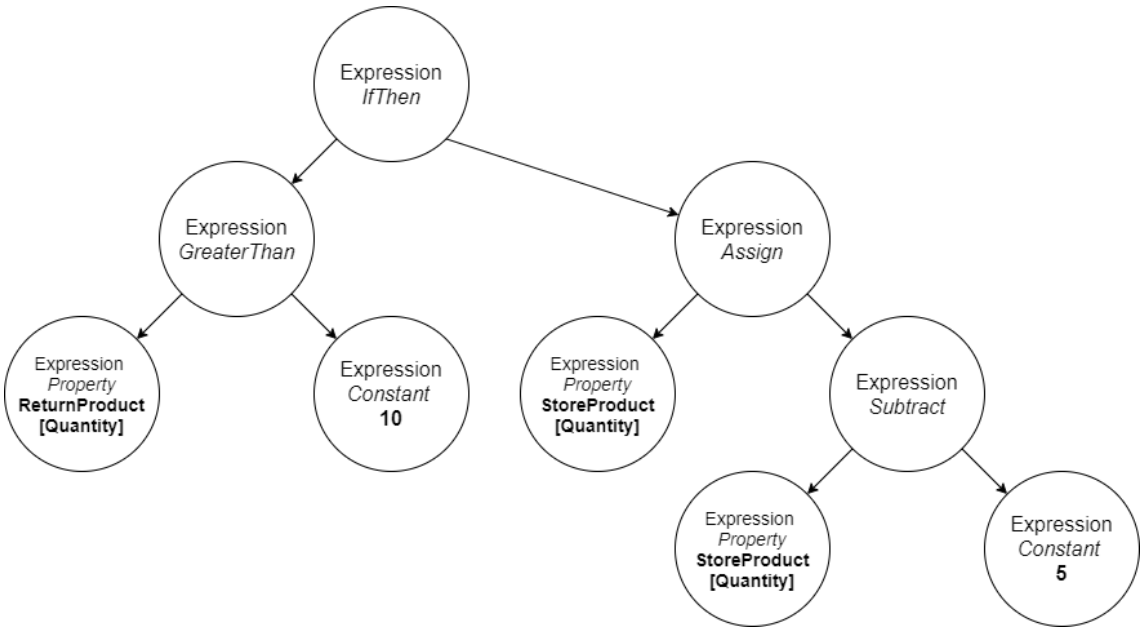
THRESHOLD_QUANTITY	**DECREMENT**
10	5

Reeglitabeli kasutamiseks on vaja teha täiendavaid muudatusi ka programmikoodis. Sellisel juhul ei muutu mitte realiseeritud valemi sisendiks olev väärtus, vaid tekib uus valem.

4.2.2 C# avaldistepuu

Avaldistepuud esitavad kompileeritavat koodi puusarnases struktuuris, kus iga lüli on avaldis. Avaldistepuus sisalduvaid avaldisi on võimalik kompileerida ja käivitada, mis võimaldab programmikoodi dünaamiliselt muuta. C# programmeerimiskeele LINQ

päringute nimeruumis on abstraktne *Expression* klass, mis esindab kõiki loogilisi meetodeid programmikoodi kirjutamiseks avaldistega [26].



Joonis 20: Uue ärinõude esitamine avaldistepuuna.

Joonis 20 kirjeldab uut ärinõuet C# avaldistepuu kaudu. Puu sõlmedes olevad avaldised käivitatakse ülevalt-alla ning vasakult-paremale.

4.2.3 Lahenduste võrdlus

Reeglitabeliga on võimalik saavutada andmetega juhitavus vastavalt nõuetele. Küll aga eeldab antud lahendus tarkvara arendamise käigus muutuvate nõuete ettenägemist. Kuigi põhiliste operatsioonide puhul on reeglina ennustatav, millisel viisil võib funktsionaalsus aja jooksul muutuda, ei pruugi see tegelikkuses alati nii olla. Downsi kirjeldatud probleem tuli esile alles aastaid pärast süsteemi arendamist [11]. Andmetega juhitavaks muutmise lisapingutused ei pruugi sel viisil arendades tagada, et hiljem muutuvad ärinõuded ei tähendaks vajadust programmikoodi täiendada.

Kirjeldatud lahendus defineerib reeglitabelid vaid nimetatud ühe äriprotsessi kohta. Mida suurem süsteem, seda enam leidub selliseid ajas muutuvaid ärilisi nõudeid, mille puhul oleks vaja tarkvara arendades kirjeldada ja arendada reeglisüsteemi. Nii võib andmemudel ühes reeglitabelitega minna mingist hetkest alates liiga suureks ja raskesti hallatavaks.

Antud töö raames on oluliseks andmetega juhitavuse omaduseks võimalus arvutada avaldiste väärtuseid vastavalt mingitele äriloogika reeglitele. Oletades, et uus ärinõue ilmnes alles tarkvara elutsükli hilisemas faasis, nõuab reeglitabelitel põhinev lahendus sellisel kujul programmikoodis valemi muudatust. Seevastu on avaldistepuudega võimalus regulatsiooniprogrammis käivitamise hetkel kompileerida ja käivitada erinevaid valemeid esitavaid avaldisi.

Nimetatud põhjustel ei lahendata antud probleemipüstitust Downsi pakutavate reeglitabelitega [11], vaid dünaamiliselt muudetavate C# avaldistepuudega [26].

4.3 Regulatsiooniprotsess avaldistepuuna

Jaotises 3.3 kirjeldati programmikoodi näitel regulatsiooni arvutuskäiku, kus tagastatud ajakirjade koguse põhjal arvutati ümber tulevikus tellitavate ajakirjade kogus. Sama protsess, mida kirjeldas Downs oma ajaveebi postituses [11] reeglitabelite abil, on seal esitatud kui vaikimisi kodeeritud programmiloogika. Käesolevas peatükis esitatakse lisandunud ärinõude “Toote koguse ümberarvutus” (vt jaotis 4.1) funktsionaalsus avaldistepuuna ning JSON formaati teisendatud sisendina. Antud JSON formaadis sisend on teenuse kasutaja võimalik sisend API lõppsõlmele. Programmikoodis käivitatakse see sisend avaldisena.

Avaldistepuu aluseks on *StoreProduct* ja *ReturnProduct* klassid ning nende atribuudid *Quantity*, mis vastavalt iseloomustavad poe tellitud toote kogust ja poe tagastatud toote kogust. Programmikoodis itereeritakse regulatsiooni käivitamisel üle poe tellitud toodete massiivi. Iga toote kohta leitakse vastav tagastatud toode ning käivitatakse avaldis. *StoreProduct* objekti atribuudi *Quantity* määramiseks on Joonis 21 esitatud näites kasutusel *Expression* abstraktsest baasklassist tuletatud meetodid *IfThen*, *GreaterThan*, *Assign*, *Property*, *Subtract* ja *Constant*. Edaspidi kasutatakse avaldistes sama baasklassi põhiseid avaldisi [27].

```
Expression.IfThen(
      Expression.GreaterThan(
            Expression.Property(returnProductParameter, "Quantity"),
            Expression.Constant(10)
      ),
      Expression.Assign(
            Expression.Property(storeProductParameter,"Quantity"),
            Expression.Subtract(
                  Expression.Property(
                        storeProductParameter, "Quantity"),
                  Expression.Constant(5)
            )
      )
);
```

Joonis 21: Avaldistepuu kujutamine C# *Expression* baasklassi meetoditega.

4.3.1 Avaldise käivitamine

Programmikoodis käivitatakse esitatud avaldis järgmiselt.

1. Luuakse uus *Expression* baasklassi kuuluv *Lambda* avaldis. Selle avaldise esimeseks sisendiks on avaldis või avaldistepuu ja teiseks sisendiks on avaldisele lisatavad argumendid.
2. *Lambda* avaldis koos sisenditega kompileeritakse.
3. Kompileeritud *Lambda* avaldis käivitatakse dünaamiliselt sisendandmetega, mis on andmebaasist loetud kirjete alusel loodud objektid.

Joonis 22 on kujutatud eelnevalt kirjeldatud avaldise käivitamist.

```
ParameterExpression returnProductParameter = Expression.Parameter(
                                    typeof(ReturnProduct), "returnProduct");
ParameterExpression storeProductParameter = Expression.Parameter(
                                    typeof(StoreProduct), "storeProduct");

var expression =
      Expression.IfThen(
            Expression.GreaterThan(
                  Expression.Property(
                        returnProductParameter, "Quantity"),
                  Expression.Constant(10)
            ),
            Expression.Assign(
                  Expression.Property(
                        storeProductParameter, "Quantity"),
                  Expression.Subtract(
                        Expression.Property(
                              storeProductParameter, "Quantity"),
                        Expression.Constant(5)
                  )
            )
     );

Expression.Lambda(expression,
      new List<ParameterExpression>
      { returnProductParameter, storeProductParameter })
.Compile().DynamicInvoke(returnProduct, storeProduct);
```

Joonis 22: Avaldise käivitamise programmikood.

4.3.2 Dünaamilised LINQ päringud avaldistena

LINQ päringukeeles kasutatakse C# avaldisi, et rakendada struktureeritud päringuid *IQueryable<T>* liidesele. C# kompilaator kompileerib kirjutatud LINQ süntaksi programmi käivitamise hetkel samuti avaldistepuuks [40]. Avaldistepuu abil on võimalik luua seega LINQ päringuid, mis tuleb kasuks kui programmi kompileerimise hetkel ei ole päringute spetsiifilised omadused ja käitumine teada.

Antud töö kontekstis on LINQ avaldiste kasutamine laiendatud võimekuseks süsteemi andmetega juhitavaks muutmisel. Varasemates näidetes on avaldistes kasutatud konstantsete väärtustena objekte, mis on LINQ päringutega andmebaasist laetud. Avaldistepuu *Linq.Expressions* [27] nimeruumi klassidega on võimalik käivitada LINQ meetodeid, mis on ühtlasi ka aluseks LINQ süntaksile.

Näitena võib siinkohal tuua poe tellitud tootele vastava tagastatud toote päringu, mis on kujutatud Joonis 23.

```
    var returnProduct =
          distribution.Return.ReturnProducts
                .FirstOrDefault(returnProduct =>
                      returnProduct.ProductId == storeProduct.ProductId);
```

Joonis 23: LINQ süntaksis tagastatud toote päring.

Antud päring ei ole dünaamiline ning süsteemi mõistes peab olema programmikoodi tasemel defineeritud. Joonis 24 on esitatud LINQ FirstOrDefault parametriseeritud päring avaldistepuuna (jaotus "*distribution*" on antud näites lihtsustamise mõttes varasemalt väärtustatud muutuja).

```
ParameterExpression returnProductParameter =
      Expression.Parameter(typeof(ReturnProduct), "returnProduct");

ParameterExpression storeProductParameter =
      Expression.Parameter(typeof(ReturnProduct), "storeProduct");

Expression.Constant(
      Expression.Call(
            typeof(Enumerable), "FirstOrDefault",
            new[] {typeof(ReturnProduct)},
            Expression.PropertyOrField(
                  Expression.Property(
                        Expression.Constant(distribution), "Return"),
                        "ReturnProducts"),
             Expression.Lambda<Func<ReturnProduct, bool>>(
                  Expression.Equal(
                        Expression.Property(
                              storeProductParameter, "ProductId")),
                        Expression.Property(
                              returnProductParameter, "ProductId")),
                  returnProductParameter)
     )
)
```

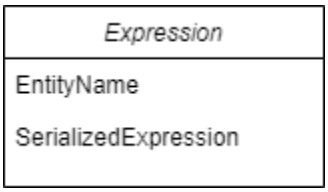
Joonis 24: LINQ *FirstOrDefault* päring avaldistepuuna.

4.3.3 API sisend

Äriloogilise otsustusprotsessi avaldistena esitamine on ainult üks samm tarkvara andmetega juhitavaks muutmisel. Selleks, et tarkvara funktsionaalsus oleks andmetega juhitav töös varasemalt kirjeldatud viisil, peab JSON formaati teisendatud avaldis olema API sisendiks. API sisendiks olevat avaldist saab programmis tagasi teisendada *Expression* tüüpi objektiks. Käesolevas projektis kasutatakse sellise teisendamise tegemiseks "*Aq.ExpressionJsonSerializer*" [41] teeki (versioon 0.18.0).

Avaldiste puhul on oluline, et kasutatakse programmi nimeruumis defineeritud klasse. Seetõttu peab pakutava lahenduse korral olema API kasutajale teada rakenduse andmebaasi skeem ja äriloogika nimeruum. Sellega kaasnevatest turvariskidest kirjutatakse jaotises 4.5.

Käesolevas projektis on kasutatud lähenemist, kus kontrolleri sisendiks aktsepteeritakse *EntityExpression* klassi objekti. Joonis 25 esitab selle klassi atribuudid.



Joonis 25: API sisendiks olevate avaldise objektide klass.

Kirjeldatava klassi sõne (stringi) tüüpi atribuudi *EntityName* väärtus viitab ORM raamistiku andmebaasi kontekstis kirjeldatud klassi nimele. Selle alusel saab koodi tasemel tüübikontrolli läbi eristada, mis tüüpi objektidele vastavat avaldist rakendatakse. Käesolevas peatükis esitatud näites on atribuudi *EntityName* väärtus “StoreProduct”. Avaldis ise on serialiseeritud kujul JSON formaadis API sisendis ning deserialiseeritakse rakenduse äriloogika kihis vastava teenuse nimeruumis.

Sarnaselt GraphQL põhimõttele, tuleb ka siin pakutud lahenduse puhul kõik päringud muuta POST tüüpi päringuteks, et päringu kehas oleks võimalik edastada avaldisi. Näitena tuuakse võrdlus jaotuse pärimise API lõppsõlmest (*endpoint*), kus on GET tüüpi kontrolleri sõlm sisendi jaoks muudetud POST tüüpi sõlmeks (Joonis 26).

```
[HttpGet("{id}")]
public IActionResult GetDistribution([FromRoute] Guid id)
{
      return Ok(_distributionService.Get(id));
}

[HttpPost("{id}")]
public IActionResult DataDrivenGetDistribution([FromRoute] Guid id,
[FromBody] EntityExpression expression)
{
      return Ok(_distributionService.Get(id, expression));
}
```

Joonis 26: GET tüüpi päring andmetega juhitavas tarkvaras.

Esitatud kaks API lõppsõlme on sama funktsionaalsust täitvad. Teine lõppsõlm erineb esimesest selle poolest, et päringu kehas saab esitada *EntityExpression* tüüpi sisendi. See sisend edastatakse äriloogika kihti, kus avaldise JSON kujul väärtus teisendatakse *Expression* tüüpi objektiks ning käivitatakse.

Antud peatükis käsitletud uuenenud äriloogika jaoks API lõppsõlme tüüpi ei muudetud, kuna tegu oli juba POST päringuga. Sarnaselt eelmisele näitele lisandub sellele lõppsõlmele sisendparameetriks *EntityExpression* (Joonis 27).

```
[HttpPost("distribution/{distributionId}")]
        public IActionResult CreateRegulation([FromRoute] Guid
distributionId, [FromBody] EntityExpression expression)
        {
            return Ok(_regulationService.Create(distributionId,
expression));
        }
```

Joonis 27: Regulatsiooni käivitamise lõppsõlm.

Punktis 4.2.1 kirjeldatud avaldise käivitamise protsessile antakse ette API poolt vastu võetud avaldis ning protsess muutub nii nagu esitab Joonis 28.

```
var deserializedExpression =
      JsonConvert.DeserializeObject<Expression>(
expression.SerializedExpression.ToString(), settings);

Expression.Lambda(
      deserializedExpression,
      new List<ParameterExpression>
            { returnProductParameter, storeProductParameter})
      .Compile().DynamicInvoke(returnProduct, storeProduct);
```

Joonis 28: Avaldise käivitamise protsess.

Kirjeldatud sisendi JSON formaadis edastamise miinuseks on sisendandmete keeruline struktuur. Joonis 29 on jaotises 4.1 esitatud lisandunud ärinõude koguse arvutamise loogika avaldistepuu JSON formaadis lühendatud kujul.

```
{
  "nodeType": "Conditional",
  "test": {
    "nodeType": "GreaterThan",
    "left": {
      "nodeType": "MemberAccess",
      "expression": {
        "nodeType": "Parameter",
        "name": "returnProduct"
      },
      "member": {
        "name": "Quantity"
      }
    },
    "right": {
      "nodeType": "Constant",
      "value": 10
    }
  },
  "ifTrue": {
    "nodeType": "Assign",
    "left": {
      "nodeType": "MemberAccess",
      "expression": {
        "nodeType": "Parameter",
        "name": "storeProduct"
      },
      "member": {
        "name": "Quantity"
      }
    },
    "right": {
      "nodeType": "Subtract",
      "left": {
        "nodeType": "MemberAccess",
        "expression": {
          "nodeType": "Parameter",
          "name": "storeProduct"
        },
        "member": {
          "name": "Quantity"
        }
      },
      "right": {
        "nodeType": "Constant",
        "value": 5
      }
    }
  },
  "ifFalse": {
    "nodeType": "Default"
  }
}
```


Joonis 29: Avaldistepuu lihtsustatud esitus JSON formaadis.

Lihtsustatud näites on igalt sõlmelt eemaldatud sellele vastava klassi tüübi ja nimeruumi definitsioonid. Sellegipoolest on näha, et isegi lihtsustatud kujul olev näide on võrdlemisi lihtsa avaldistepuu kirjeldamiseks mahukas ja keerukas. Täispikk näide selle avaldistepuu JSON kujust on leitav Lisas 1.

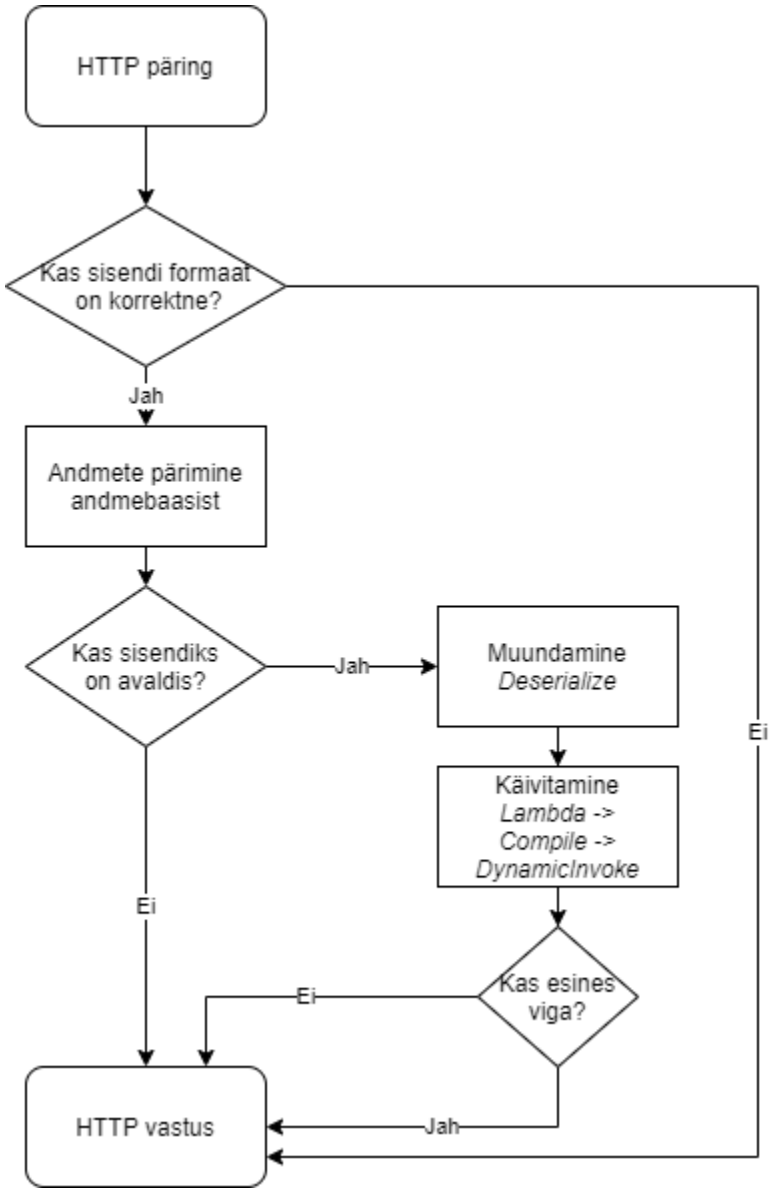
Antud töö raames ei looda avaldiste koostamiseks graafilist liidest, kuna Visual Studio programmeerimiskeskkonna võimekus avaldiste kirjutamiseks on väga hea. Näide kirjutatud avaldisest Visual Studio keskkonnas koos keskkonna soovituste süsteemiga on kuvatõmmistena esitatud Lisades 2 ja 3. Avaldised kirjutatakse Visual Studio keskkonnas valmis ning teisendatakse JSON formaati kasutades "*Aq.ExpressionJsonSerializer*" tarkvarateegi *Serialize* meetodit. Teisendamise tulemuseks olevat JSON objekti saab kasutada edaspidi API sisendina.

4.4 Rakenduse töövoog

Oluline on, et andmetega juhtimiseks vajalikud muudatused ei mõjutaks algset funktsionaalsust. See tähendab, et kui mitmevalduslikus süsteemis kasutab üks klient avaldiste võimekust äriloogika muutmiseks, siis ei tohi see kuidagi mõjutada teiste klientide jaoks mõeldud funktsionaalsust. Samuti ei tohiks avaldistes esinevad vead põhjustada tõrkeid konkreetsele klientsüsteemile, mille esitatud teenusenõudesse oli avaldis kaasa pandud. Süsteemi töövoo ja struktureeritud järjekorra alusel on võimalik alati tagavaravariandina tagada tarkvara algse funktsionaalsuse toimimine. Käesolevas jaotises kirjeldatakse avaldistega juhitavate teenuste programmi töövood.

4.4.1 Andmepäringud

Andmepäringute töövoogu andmetega juhitavas rakenduses kirjeldab Joonis 30.



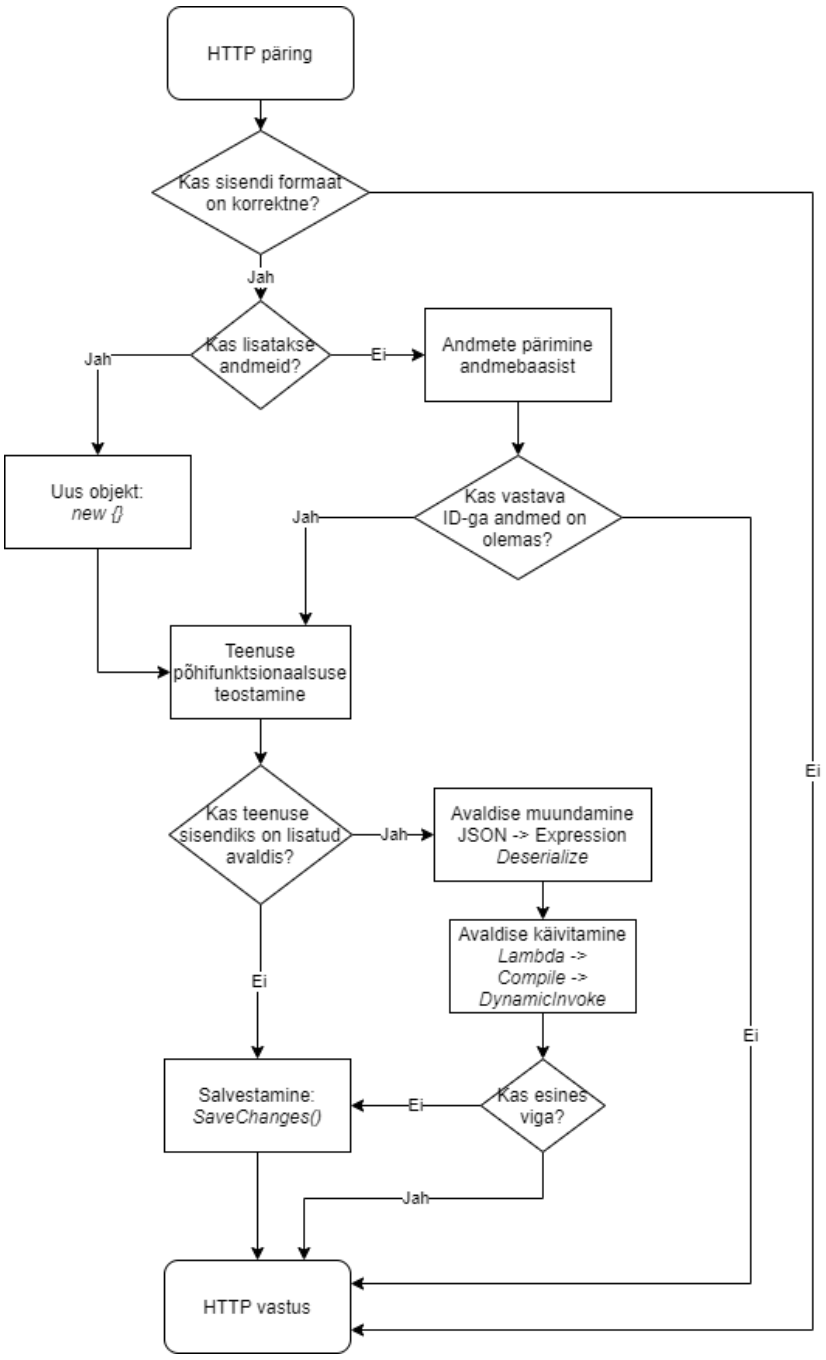
Joonis 30: Andmete pärimise töövoog.

Joonis 30 on kujutatud API andmete pärimise töövoog, kus täiendava lisategevusena tehakse andmete muundamine sisendiks oleva avaldise põhjal. API päringu käivitamine ilma spetsiifilise avaldise sisendita käivitab vaikimisi funktsionaalsuse ning tagastab andmed. Selline töövoog garanteerib, et põhifunktsionaalsust vajavatel klientsüsteemidel liidese integratsioon ja funktsionaalsus ei muutu.

4.4.2 Andmemuudatused

Andmemuudatusi teostavate operatsioonide puhul on võimalik eristada kahte erinevat töövoogu: täielikku andmetega juhtimist ja täiendavat andmetega juhtimist.

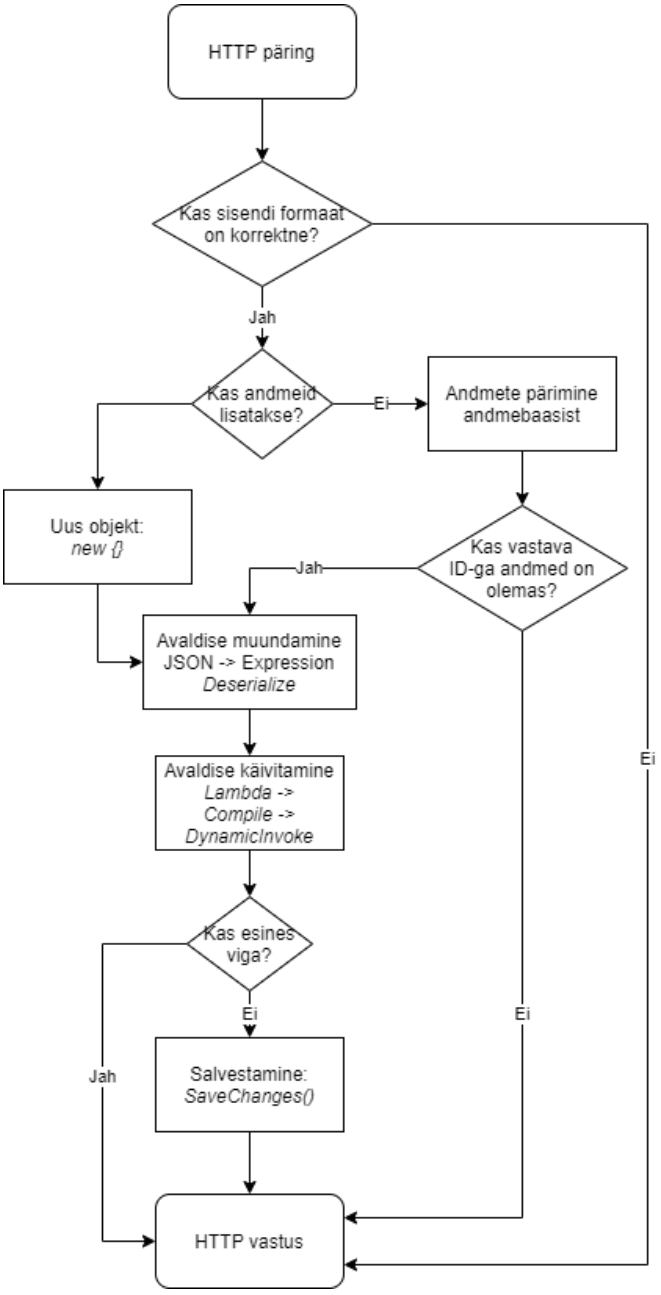
Täiendava andmetega juhtimise töövoo korral (Joonis 31) käivitatakse rakenduse vaikimisi realiseeritud äriloogika ja sellele *lisaks* etteantud avaldis. Selles peatükis esitatud uuenenud äriloogika nõude puhul tähendab täiendav andmetega juhtimine, et regulatsiooni töövoogu käivitades rakendatakse sisseehitatud regulatsiooniloogika ja seejärel käivitatakse uus regulatsioon, mis tuleneb avaldisest:



Joonis 31: Täiendavalt andmetega juhitav töövoog.

Avaldise puudumise korral on tegemist tavapärase POST või UPDATE tüüpi päringuga. Sellise päringu puhul käivitub ainult rakendusse sisse ehitatud äriloogika.

Täieliku andmetega juhtimise puhul (Joonis 32) rakendusse sisseehitatud äriloogikat ei rakendata ning peale uue rea tabelisse lisamist või olemasoleva rea väljade uuendamist käivitub vaid päringusse ette antud avaldis. Kui sellise päringu puhul avaldis sisendist puudub, siis tehakse ainult rea lisamine või uuendamine:



Joonis 32: Täielikult andmetega juhitav töövoog.

Mõlema töövoo puhul on oluline, et andmebaasis salvestatakse muudatused vaid siis, kui avaldiste käivitamises ei esine vigu või kui sisendis ei olnud avaldisi. See on oluline vältimaks andmekvaliteedi vigu. Avaldise käivitamise vea korral tagastatakse vastav HTTP vastus.

Täiendava või täieliku andmetega juhitavuse valikut on võimalik API klientidele võimaldada kas läbi API sisendiks oleva tõeväärtustüüpi väärtuse või erinevate API lõppsõlmedega. Üksiku vastutuse ja otstarbe lahususe printsiipide [42] seisukohalt oleks erinevad API lõppsõlmed eelistatud. Antud töös realiseeritud regulatsiooni käivitamise lõppsõlm realiseerib täielikku andmetega juhitavust.

4.5 Turvalisus

API sisendisse antud avaldistega äriloogiliste protsesside juhtimine tähendab tarkvara ja andmete turvalisuse tagamiseks lisandunud võimalikke arhitektuurilisi kitsendusi.

Andmetega juhtimine API sisendandmetega tekitab kindlasti küsimuse, et kas rakendus on haavatav läbi SQL süstimise (*injection*). Pakutud lahenduses on kasutatud Entity Framework teeki ja LINQ päringukeelt andmebaasiühenduseks ja päringute koostamiseks. Kuigi LINQ avaldiste põhjal koostatakse käivitamiseks SQL päringud, ei ole antud viisil esitatud lahendus SQL süstimise kaudu haavatav. Kasutaja sisendi põhjal koostatakse LINQ avaldised, kuid ei kasutata *FromRawSql* meetodit – st LINQ päringute sisendiks ei ole kunagi puhas SQL kood [43].

Lahenduse arhitektuurilised valikud võivad seevastu küll pakkuda võimalusi andmete varguseks. Kuna avaldiste kaudu on võimalik käivitada ja koostada LINQ päringuid, siis peab mitmevalduslikus süsteemis jälgima, et teiste valdajate andmed ei oleks avaldiste kaudu ligipääsetavad. Entity Framework klasside omavahelisi suhteid teades ja LINQ päringute abil on võimalik avaldiste kaudu pärida ka seotud andmeid. Selliselt kui andmebaas on üles ehitatud nii, et mitme valdaja andmed on samas andmebaasis ning need andmed on seotud ka ühiskasutatavate andmetega, nagu näiteks klassifikaatorid, on avaldiste ja ühiste seotud andmete kaudu võimalik saada ligipääs ka võõrastele andmetele.

Efektiivne viis ennetada andmevargusi dünaamiliste LINQ päringute abil oleks läbi mitmevaldusliku arhitektuuri, kus igal valdajal on oma andmebaas. Nii on kõik LINQ päringutest koostatud SQL päringud käivitatud konkreetse valdaja andmebaasis ning puudub ligipääs võõrastele andmetele. Eraldi andmebaasidega mitmevalduslikus mudelis on valdaja andmete hoidmiseks ka jagatud andmebaas (vt Joonis 13). Andmetega

juhitavust selle andmebaasiga seotud funktsionaalsuse raames realiseerida ei tohi. Jagatud andmebaasi eesmärk peab süsteemis olema vaid valdajate ligipääsuandmete hoidmiseks.

Alternatiivselt on võimalik tagada andmevarguste vastu täiendavat turvalisust ka jagatud andmebaasiga süsteemis. Kasutades andmebaasisüsteemis ridade tasemel defineeritud poliitikaid on võimalik piirata, millised kasutajad (valdajad) millistele tingimustele vastavaid ridu tabelites tohivad lugeda ja muuta. Tabelile on vaja kirjeldada andmebaasi funktsioon, milles kontrollitakse valdaja ID vastavust päringu tegijaga. Seejärel on vaja luua turvapoliitika, kus kasutatakse seda funktsiooni filtrina [44]. Iga kord, kui mistahes viisil üritatakse tabelist andmeid lugeda või muuta rakendab andmebaasisüsteem poliitikast tulenevat filtrit. Nii ei saa sellest poliitikast mööda minna (erinevalt rakenduse tasemel defineeritud poliitikast, millest saab mööda minna andmeid ilma rakendust kasutamata lugedes). Entity Framework raamistiku tasemel on samuti võimalik kirjeldada sarnane filtrisüsteem, kuid juba rakenduse koodi tasemel [45].

API kliendi poolelt kujutab valdaja andmetele ohtu olukord, kus valdajapoolse süsteemi lõppkasutajal on ligipääs API sisendiks kaasa antavale avaldisele. Kuna valdaja defineerib enda süsteemis avaldise, mis on teenuse sisendiks, on selle valdaja vastutusalas ka tagada, et tema kliendid omakorda ei saaks seda avaldist kuritarvitada. Näitena võib tuua soodustuse arvutamise. Valdaja on defineerinud avaldise, millega mingi perioodi vältel rakendatakse toodete hinnale soodustust mingi protsendi ulatuses. Avaldises korrutatakse toote hind läbi vastava koefitsiendiga ning API tagastab arvutatud soodustusega tootehinna. Kui lõppkasutajal on ligipääs selle avaldise muutmisele, on võimalik seda koefitsienti vähendada ja seeläbi tekitada endale kasu valdaja arvelt. API mõistes ei ole sellist rünnakud võimalik kuidagi peatada. Teenus eeldab, et valdaja on koostanud avaldise ise ning seda pole kolmandad osapooled muutnud.

LINQ dünaamilised päringud pakuvad andmetega juhitavuse mõistes laiendatud võimekust ja õigustavad lisapingutusi andmetega juhitava tarkvara arendamisel. Teisalt tähendab selline omadus, et andmebaasi struktuur oleks teada ka API kasutajale. Kahju tekitamise eesmärgil rünnakut planeerivatel osapooltel on selle teadmisega võimalus andmevargusi paremini planeerida. Lahenduseks võib avaldiste aluseks võtta ainult andmebaasist programmi mällu laetud andmed. Selliselt on andmetega juhitavusest tulenev võimekus suuresti piiratud – andmetega saab juhtida vaid API väljundiks olevat objekti, mitte äriloogikat. Lisaks ütleb Shannoni maksiim [46], et süsteem tuleb disainida

eeldusel, et vaenlane tunneb seda süsteemi. Turvalisus ei tohi rajaneda eeldusele, et süsteemi ülesehitus on ründaja ees saladus. Kuna andmebaas koos seal hoitud andmetega teenuste mõistes valdaja oma, ei ole ilmtingimata halb, kui valdajapoolse süsteemi arendaja tunneb vastavat andmebaasi mudelit.

4.6 Jõudlus

API andmetega juhitavus töös kirjeldatud viisil tähendab iga protsessi keerukuse suurenemist, millest tulenevalt võivad tekkida jõudlusprobleemid. Käesolevas jaotises koostatakse esitatud näiteprotsessile jõudlustestid ning hinnatakse selle põhjal lahenduse mõju jõudlusele. Testide koostamiseks kasutatakse Apache JMeter [47] rakenduse versiooni 5.2.1.

4.6.1 Jõudlustestid

Koormustestide kaudu on võimalik kontrollida süsteemi töökindlust ning leida koormuspunkt, mille puhul tarkvara või teenus enam kõikidele päringutele vastata ei jõua. Koormustestimine võimaldab tuvastada teenuste vastamisaega ja selle muutumist erineva tasemega koormuste korral, tarkvara disainist tulenevaid probleeme, tarkvara konfiguratsioonist tulenevaid puuduseid või ka riistvaralisi limiite [48].

Antud töö mõistes on koormustestide eesmärk tuvastada disainilahenduse probleeme, eelkõige andmetega juhitava lahenduse võimalikku vähenenud jõudlust ja pikenenud vastamisaega võrreldes põhifunktsionaalsust realiseeriva teenusega. Koormustestidega hinnatakse, milliselt mõjutab tööprotsessi kiirust sisendandmetena avaldise vastuvõtmine, selle muundamine *Expression* tüüpi avaldistepuuks ning käivitamine programmikoodis.

Jõudlustestide jaoks kohandati antud töös esitatud regulatsiooni loomise teenuse äriloogikat selliselt, et selle põhifunktsionaalsus oleks sama kasutusjuhus UC-4 kirjeldatud loogikale. Põhifunktsionaalsus on ümber kirjutatud selleks, et API põhifunktsionaalsus ja täielikult andmetega juhitav API lõppsõlm tagastaks samade lähteandmete korral samad tulemused. Selliselt on võimalik koormustestidega hinnata vaid andmetega juhitavusest tulenevate erisuste mõju teenuse koormustaluvusele.

Jõudlustestid käivitatakse mitmel koormusastmel: üks päring sekundis, sada päringut sekundis, tuhat päringut sekundis ja viis tuhat päringut sekundis. Kuna andmetega juhitava teenuse korral on päringu kehas sisendina JSON kujul avaldis, mis muundatakse äriloogika kihis ja käivitatakse, siis eeldatakse, et kõikide koormuste korral on andmetega juhitava päringu töötlemisele kuluv koguaeg pikem kui põhifunktsionaalsust realiseerival teenusel.

Jõudlustestid käivitatakse jaotusele, kus on tellitud Joonis 33 kujutatud tooted vastavate kogustega.


```
"DistributionProducts": [
        {
            "Id": "51eaf777-0b37-4baf-b160-253f368a6a0f",
            "ProductId": "369e4b1e-1f96-4457-8aa1-308cf2f91144",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 400
        },
        {
            "Id": "5625b3cd-72cb-4dbc-b261-58160c225e0b",
            "ProductId": "1239c984-9579-43a9-8aeb-02c50cbe970f",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 200
        },
        {
            "Id": "596799cc-eb37-4a96-bc46-6af6d13b011b",
            "ProductId": "4e26d63c-18f7-4240-a53d-31f5a67a499a",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 50
        }
    ]
```


Joonis 33: Koormustestide jaotatud toodete lähteandmed JSON kujul.

Jaotusele vastav tagastatud toodete hulk on esitatud Joonis 34.

```
"ReturnProducts": [
        {
            "Id": "c3044a99-678c-448e-98ae-08d7f1cbe968",
            "Quantity": 150,
            "ProductId": "369e4b1e-1f96-4457-8aa1-308cf2f91144",
            "ReturnId": "1d68c797-7320-4b0f-a90a-08d7f1cbe966"
        },
        {
             "Id": "9d9cf82e-313d-4c5c-98af-08d7f1cbe968",
             "Quantity": 200,
             "ProductId": "1239c984-9579-43a9-8aeb-02c50cbe970f",
             "ReturnId": "1d68c797-7320-4b0f-a90a-08d7f1cbe966"
         },
         {
              "Id": "4289117e-3aee-4ac2-98b0-08d7f1cbe968",
              "Quantity": 0,
              "ProductId": "4e26d63c-18f7-4240-a53d-31f5a67a499a",
              "ReturnId": "1d68c797-7320-4b0f-a90a-08d7f1cbe966"
          }
    ]
```


Joonis 34: Koormustestide tagastatud toodete lähteandmed JSON kujul.

Regulatsiooni käivitamise tulemusena eeldatakse, et API tagastab muudetud kogustega poe tellitud toodete massiivi, mis on esitatud Joonis 35.

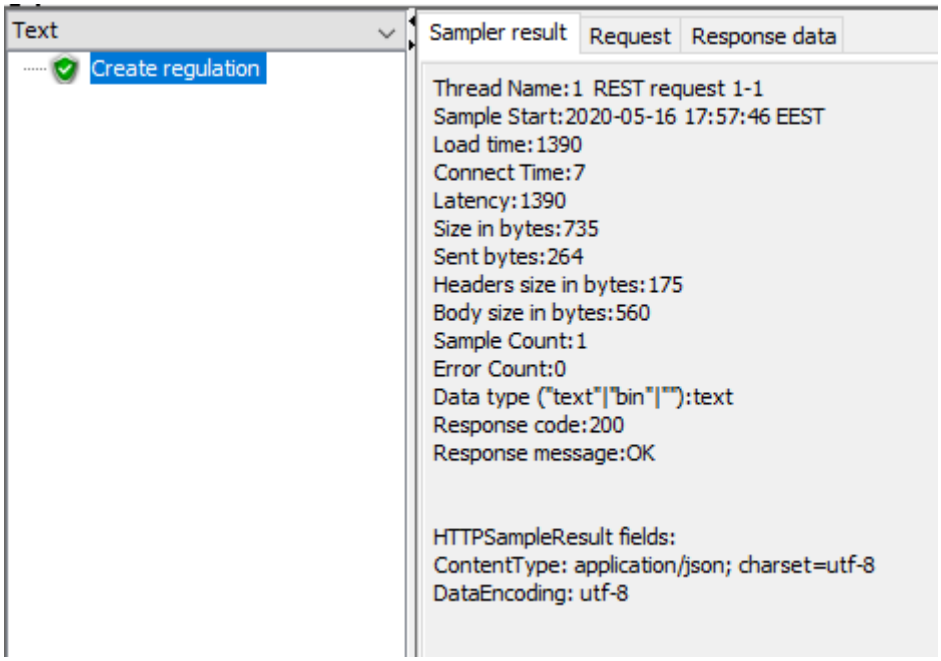

```
"StoreProducts": [
        {
            "Id": "51eaf777-0b37-4baf-b160-253f368a6a0f",
            "ProductId": "369e4b1e-1f96-4457-8aa1-308cf2f91144",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 395
        },
        {
            "Id": "5625b3cd-72cb-4dbc-b261-58160c225e0b",
            "ProductId": "1239c984-9579-43a9-8aeb-02c50cbe970f",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 195
        },
        {
            "Id": "596799cc-eb37-4a96-bc46-6af6d13b011b",
            "ProductId": "4e26d63c-18f7-4240-a53d-31f5a67a499a",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 50
        }
    ]
```


Joonis 35: Koormustestide reguleeritud poe tellitud toodete kogus JSON kujul.

4.6.2 Regulatsiooni REST API jõudlustestid

Siin jaotises testitakse APIt, mis ei ole andmetega juhitav. Regulatsiooni API lõppsõlme käivitatakse neljas faasis. Ajaline vahemik, mil päringud käivitatakse, on igal testil üks sekund – päringud saadetakse selle aja jooksul võrdselt hajutatult. Igas faasis suurendatakse päringute arvu ehk koormust.

Üksiku saadetud API päringu korral on kulunud aeg päringu saatmisest vastuse saamiseni 1360 millisekundit. Saadetud päringu vastuse andmeid ja staatust kujutatakse Joonis 36.



Joonis 36: REST API koormustesti päringu staatus.

Saja päringu saatmise korral peab tarkvara koormusele vastu väga hästi. Võrdluses ühe saadetud päringuga pikeneb keskmine vastuse laadimisaeg 1489 millisekundini. Muutus nii väikese koormuse korral on veel väga väike. Summeeritud tulemused 100 päringu saatmisest on kujutatud Joonis 37.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Create regulat...	100	1489	1009	2009	296.06	0.00%	49.3/sec	35.38	12.71	735.0
TOTAL	100	1489	1009	2009	296.06	0.00%	49.3/sec	35.38	12.71	735.0

Joonis 37: REST API koormustesti 100 päringu summeeritud tulemused.

Päringute arvu suurenemine tuhande päringuni pikendab keskmist vastamisaega juba rohkem, kuna päringute saatmise hetkel tekib API kontrollerile järjekord. Hilisemalt töödeldavate päringute vastamisaeg pikeneb seega ka järjekorras ooteaja võrra. Keskmine vastamisaeg tuhande saadetud päringu korral on 2782 millisekundit. Summeeritud tulemused 1000 päringu saatmisest on kujutatud Joonis 38.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Create regulation	1000	2782	1822	4315	812.10	0.00%	222.7/sec	159.86	57.42	735.0
TOTAL	1000	2782	1822	4315	812.10	0.00%	222.7/sec	159.86	57.42	735.0

Joonis 38: REST API koormustesti 1000 päringu summeeritud tulemused.

Viie tuhande päringu saatmise korral saavutatakse arenduskeskkonnas teenuse koormuse limiit. Liiga suure päringute arvu tõttu ei suuda rakendus vastata 10,28% päringutest ning tagastab vea. Rakenduse piirangud ei võimalda töödelda korraga nii suurt hulka ning tagastatakse päringulimiidi ületamise piirang. See on nii lühikesel ajahetkel suure hulga päringute puhul ootuspärane ning aktsepteeritav tulemus. Keskmine vastamisaeg pikenes 12850 millisekundini. Summeeritud tulemused 5000 päringu saatmisest on kujutatud Joonis 39.

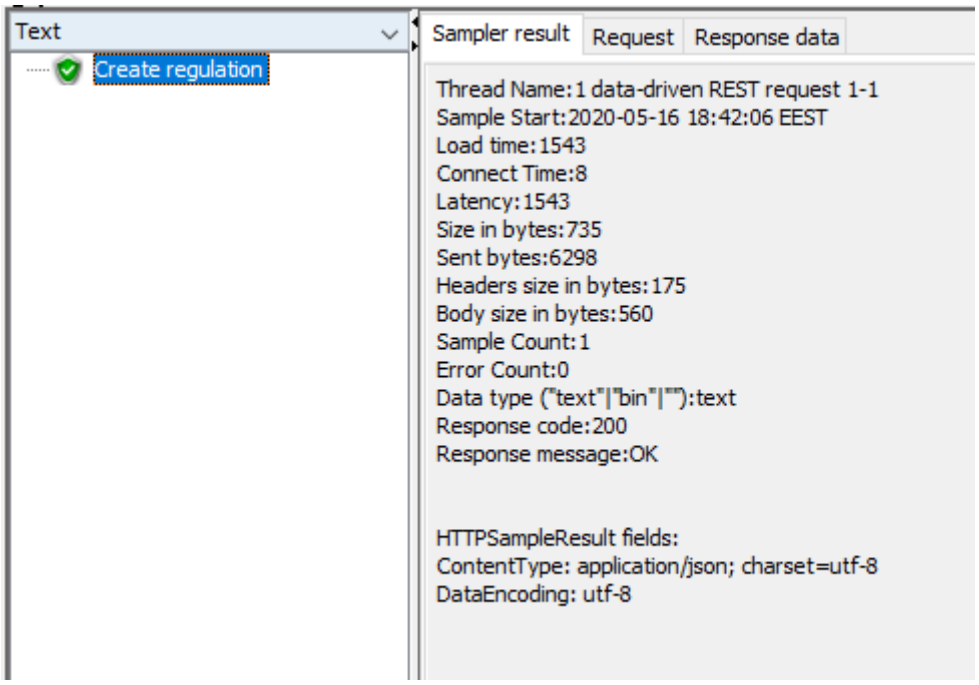
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Create regulation	5000	12850	8785	14307	1088.48	10.28%	288.3/sec	271.39	66.69	963.8
TOTAL	5000	12850	8785	14307	1088.48	10.28%	288.3/sec	271.39	66.69	963.8

Joonis 39: REST API koormustesti 5000 päringu summeeritud tulemused.

4.6.3 Andmetega juhitava API jõudlustestid

Andmetega juhitava teenuse API päringute korral rakendatakse samu lähteandmeid, mida rakendati põhifunktsionaalsust esitava API koormustestidel. Erinevuseks on sisendandmed. Andmetega juhitava teenuse sisendiks on JSON formaadis avaldis, mis on leitav Lisas 1.

Üksikult saadetud API päringu korral on vastamisaeg 1543 millisekundit. Päringu vastuse andmed ja staatus on kujutatud Joonis 40.



Joonis 40: Andmetega juhitava REST API koormustesti päringu staatus.

Saja päringu korral on tulemused ootuspäraselt väikesel määral suurenenud. Teenus suudab sellist hulka päringuid töödelda väga hästi ning keskmine päringu töötlemisaeg suureneb 1571 millisekundini. Joonis 41 on esitatud koormustesti summeeritud tulemused.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
Create regul...	100	1571	1091	2095	295.02	0.00%	47.3/sec	33.94	290.80	735.0
TOTAL	100	1571	1091	2095	295.02	0.00%	47.3/sec	33.94	290.80	735.0

Joonis 41: Andmetega juhitava REST API koormustesti 100 päringu summeeritud tulemused.

Päringute arvu suurenemisel tuhande päringuni ei saavutata ka andmetega juhitava API puhul punkti, kus rakendus ei suudaks koormusega toime tulla. Keskmine vastamisaeg pikeneb 3920 millisekundini. Summeeritud tulemused tuhande päringuga koormustestist on esitatud Joonis 42.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
Create regul...	1000	3920	3370	5115	448.08	0.00%	195.2/sec	140.08	1200.31	735.0
TOTAL	1000	3920	3370	5115	448.08	0.00%	195.2/sec	140.08	1200.31	735.0

Joonis 42: Andmetega juhitava REST API koormustesti 1000 päringu summeeritud tulemused.

Viie tuhande päringu saatmise korral on andmetega juhitava API murdumispunkt ootuspäraselt varasem, kuna päringute kehas saadetud baitide hulk on oluliselt suurem kui andmetega juhitava päringu korral. Kuna iga päringu vastuvõtmiseks ja töötlemiseks kulub teenuse poolel rohkem ressurssi, on ka vastuse saavate päringute osa pisut väiksem. Samaaegsete päringute limiidi ületamise vastuse said 15,28% päringutest ning keskmine vastuse aeg on 20 855 millisekundit. Summeeritud koormustestide tulemused on esitatud Joonis 43.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
Create regul...	5000	20855	9415	26946	4991.70	15.28%	177.8/sec	186.82	926.58	1076.0
TOTAL	5000	20855	9415	26946	4991.70	15.28%	177.8/sec	186.82	926.58	1076.0

Joonis 43: Andmetega juhitava REST API koormustesti 5000 päringu summeeritud tulemused.

4.6.4 Testitulemuste võrdlus

Andmetega juhitava API puhul on sisendandmete hulk ning JSON formaadis avaldiste muundamiseks ja käivitamiseks kuluv aeg suurenenud. Sellegipoolest on koormustestide tulemustest näha, et API vastuse aeg erineb põhifunktsionaalsust esitava API ja andmetega juhitava API puhul võrdlemisi vähe. Teenuste koormuse suurendamisel on

aga näha, et mida enam kulub aega päringu vastuvõtmisele ja töötlemisele, seda enam suureneb ka järjekorras ootavate päringute aeg ning väheneb koormustaluvus.

Koormustestide aluseks võetud funktsionaalse näite puhul on toodete koguse ümberarvutamise avaldis võrdlemisi lihtne ning ka andmetega juhitav API ei vaja väga palju aega sisendi töötlemiseks ja käivitamiseks. Samas on koormustestide puhul kõige suurema API koormuse puhul näha, et vastusena samaaegsete päringute limiidi ületamise veateate saanud päringute osa suureneb andmetega juhitava API puhul viis protsenti võrreldes tavalise REST API koormustestide tulemustega. Sellest on võimalik omakorda järeldada, et avaldiste keerukuse suurenemisel ja käivitamisele kuluva aja pikenemisel võib rakenduse koormustaluvus veelgi enam väheneda.

Kui sisendandmetest tulevad avaldised on töötlemiseks/täitmiseks liiga ajakulukad või jäävad tsüklisse, siis tuleb avaldise käivitamine peatada ja tagastada HTTP tulemuseks veateade. Vastasel juhul võib päringute töötlemiseks kuluda keerukate avaldiste tõttu väga palju riistvara jõudlust ja aega ning selle tõttu võib järjekorras ootavate päringute täitmine venida liiga pikaks või jääda vastuseta. Kui C# programmeerimiskeele tasemel seatud ajalimiidid on andmetega juhitava teenuse jaoks liiga pikad, siis saab avaldise käivitamisele konfigureerida ajapiirangu. Joonis 44 on näidatud, kuidas avaldise käivitamisele on lisatud ajalimiidiks kakskümmend viis sekundit. Seda ajalimiiti on võimalik defineerida nii koodi tasemel kui ka arenduse käigus lisada rakenduse konfiguratsioonifaili üldise avaldiste käivitamise ajalimiidina või mõne kindla operatsiooni põhise limiidina.

```
var task = Task.Run(() => Expression.Lambda(deserializedExpression,
                          new List<ParameterExpression>
{returnProductParameter, storeProductParameter}).Compile()
                     .DynamicInvoke(returnProduct, storeProduct));

if (!task.Wait(TimeSpan.FromMilliseconds(25000)))
      throw new TimeoutException();
```

Joonis 44: Avaldise käivitamine ajalimiidiga C# programmikoodis

Kokkuvõtvalt võib koormustestide põhjal öelda, et sama funktsionaalsuse realiseerimine programmikoodis ja täielikult andmetega juhitava API kaudu ei ole koormustaluvuse ja päringu töötlemisele kuluva aja osas väga suurel määral erinevad. Sellegipoolest kujutab andmetega juhitav teenus suurenenud riski jõudluse vähenemiseks. Sisendina teenuse klientidelt tulevaid avaldisi ei saa teenusehaldaja kontrollida ning seetõttu on mõistlik

seada programmikoodis avaldistele ajalised piirangud. Ajapiirangu ületanud avaldiste korral on mõistlik teenusepakkujal kontrollida, kas esitatud avaldis on optimaalselt koostatud. Samuti on väga keeruliste protsesside korral võimalik teenusepakkujal suurendada avaldise käivitamiseks lubatud ajalimiiti või kaaluda täienduse tegemist teenuse põhifunktsionaalsusesse. Pahatahtlike rünnakute vastu, kus teenuse töös tõrgete põhjustamiseks antakse sellele meelega ette liiga keerulisi avaldisi, on võimalik korraldada kaitset teenuse tasemel. Korduvalt keerulisi ja ajalimiidi ületavaid päringuid saatva kliendi päringuid on võimalik takistada. Takistada saab konkreetse sisendiga päringuid, kui tegu on vaid üksikute kordadega. Seda on võimalik teha pidades teenuse tasemel meeles vigase päringu sisend. Sama sisendiga päringule saab edaspidi tagastada veateate. Ilmselgelt rünnakule viitava päringute koguse korral on võimalik kõik konkreetse kliendi päringud süsteemile täielikult peatada.

4.7 Tulemuste hindamine

Andmetega juhitava teenuse puhul on oluline, et teenuse põhifunktsionaalsus töötaks ning selle funktsionaalsuse omadused oleksid sisendandmete kaudu muudetavad. Peatükis 4 lahendati uus äriloogika nõue “Toote koguse ümberarvutus” andmetega juhitavalt.

Lisandunud äriloogiline nõue reguleerib sarnaselt põhifunktsionaalsusele toote kogust järgmiseks jaotuseks. Selleks, et toote koguse regulatsiooni ei käivitataks kahel korral järjest, rakendati täielikku andmetega juhitavust. Teiste sõnadega, peatükis 3 kirjeldatud kasutusjuhule UC-3 vastav tarkvara tehti ümber selliselt, et toodete jaotuse loogika (olgu see selline nagu see oli algselt peatükis 3 kirjeldatud või selline nagu näeb ette peatükis 4 tekkinud uus nõue) saab vastavale teenusele ette anda.

4.7.1 Toote koguse vähendamine

Andmetega juhitavuse testimiseks käivitatakse regulatsiooniprotsess. Regulatsiooni esitav avaldis kontrollib, kas tagastatud toote hulk on suurem kui 10. Kui toote hulk ületab lävendi, siis vähendatakse järgmiseks jaotuseks toote hulka 5 ühiku võrra. Regulatsiooni algandmed on esitatud Joonis 45.

1. Tellitud ja viimase jaotusega edasimüüjale toimetatud tooted:

```
"DistributionProducts": [
        {
            "Id": "51eaf777-0b37-4baf-b160-253f368a6a0f",
            "ProductId": "369e4b1e-1f96-4457-8aa1-308cf2f91144",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 400
        },
        {
            "Id": "5625b3cd-72cb-4dbc-b261-58160c225e0b",
            "ProductId": "1239c984-9579-43a9-8aeb-02c50cbe970f",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 200
        },
        {
            "Id": "596799cc-eb37-4a96-bc46-6af6d13b011b",
            "ProductId": "4e26d63c-18f7-4240-a53d-31f5a67a499a",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 50
        }
    ]
```


Joonis 45: Tellitud tooted enne regulatsiooni.

2. Viimase jaotuse järel tagastatud tooted (Joonis 46).


```
"ReturnProducts": [
        {
            "Id": "c3044a99-678c-448e-98ae-08d7f1cbe968",
            "Quantity": 150,
            "ProductId": "369e4b1e-1f96-4457-8aa1-308cf2f91144",
            "ReturnId": "1d68c797-7320-4b0f-a90a-08d7f1cbe966"
        },
        {
            "Id": "9d9cf82e-313d-4c5c-98af-08d7f1cbe968",
            "Quantity": 200,
            "ProductId": "1239c984-9579-43a9-8aeb-02c50cbe970f",
            "ReturnId": "1d68c797-7320-4b0f-a90a-08d7f1cbe966"
        },
        {
            "Id": "4289117e-3aee-4ac2-98b0-08d7f1cbe968",
            "Quantity": 0,
            "ProductId": "4e26d63c-18f7-4240-a53d-31f5a67a499a",
            "ReturnId": "1d68c797-7320-4b0f-a90a-08d7f1cbe966"
        }
    ]
```


Joonis 46: Tagastatud tooted.

Regulatsiooni käivitamise tulemusena oodatakse, et esimese kahe toote kogused oleks vähenenud viie ühiku võrra (400 => 395 ja 200 => 195). Viimase toote kogust avaldis muuta ei tohi, kuna tagastati null toodet. Regulatsiooni tulemusel on toodete kogused muutunud jaotuseks nagu kirjeldab Joonis 47.

```
"StoreProducts": [
        {
            "Id": "51eaf777-0b37-4baf-b160-253f368a6a0f",
            "ProductId": "369e4b1e-1f96-4457-8aa1-308cf2f91144",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 395
        },
        {
            "Id": "5625b3cd-72cb-4dbc-b261-58160c225e0b",
            "ProductId": "1239c984-9579-43a9-8aeb-02c50cbe970f",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 195
        },
        {
            "Id": "596799cc-eb37-4a96-bc46-6af6d13b011b",
            "ProductId": "4e26d63c-18f7-4240-a53d-31f5a67a499a",
            "StoreId": "62eef5ae-0c22-4fc1-aa79-4124ab298eec",
            "Quantity": 50
        }
    ]
```


Joonis 47: Tellitud tooted pärast regulatsiooni.

Protsessi käivitamise tulemuste põhjal võib kinnitada, et teenus on andmetega juhitav esitatud nõuetele vastaval moel.

4.7.2 Andmetega juhitava tarkvara „halvad lõhnad“

Nagu jaotise 4.4.1 tulemuste pealt on näha, siis andmetega juhitavate operatsioonide kaudu on võimalik muuta funktsionaalsust, mille esialgset dünaamilisust analüüsi (nõuete kogumise) käigus ei kirjeldatud. Samas on kirjeldatud viisil arendatud programmikoodil ja programmi juhtimisel ka omad negatiivsed küljed ehk “halvad lõhnad” [4] [5], mida järgnevalt nimetatakse.

1. Keeruline sisend – käesoleva töö pakutud lahendus ei paku avaldiste esitamiseks graafilist liidest. Kirjeldatud näidete põhjal on ka lihtsamate avaldiste JSON kujul sisend väga suur ning raskesti loetav (vt Joonis 29).
2. Sisendi genereerimine – lisaks keerulisele sisendi JSON struktuurile, ei ole antud lahenduses ka mugavat viisi nimetatud sisendi loomiseks. Sel kujul sisendi käsitsi kirjutamine on väga keeruline – graafilise liidese puudumisel on parimaks alternatiiviks kirjutada avaldis Visual Studio keskkonnas ning muundada JSON kujule. Näited Visual Studio keskkonna kasutamisest on kuvatõmmistena esitatud Lisades 2 ja 3.
3. Andmemudel on klientidele avalik – kui teenuse eesmärk on klientidele pakkuda võimalust funktsionaalsuse kohandamiseks, siis peab klient tundma andmebaasi

struktuuri. See võib olla turvarisk. Teisalt ütleb Shannoni maksiim [46], et süsteem tuleb disainida eeldusel, et vaenlane tunneb seda süsteemi, ehk teiste sõnadega süsteemi turvalisus ei tohiks rajaneda eeldusele, et süsteemi ülesehitus on võimaliku ründaja ees saladus.

4. "Surnud kood" (*Dead code* [49]) – arendusprotsessi käigus tuleb lisada kõikidele või vähemalt olulisematele äriloogilistele protsessidele sisendi muundamise ja avaldiste käivitamise funktsionaalsus. Kui sisendiks ei saadeta avaldisi, on tegu "surnud koodiga", mis iga kord kontrollib avaldise olemasolu, kuid terve avaldise käivitamise voog jääb käivitamata.

Nimetatud probleemidest esimesele kolmele oleks kõige sobivam lahendus graafiline liides avaldiste loomiseks. Selline "liivakasti" keskkond oleks lüli andmebaasiskeemi ja liidese kasutaja vahel. Selle liidese kaudu oleks võimalik koostada soovituste süsteemi alusel avaldisi ning käivitada neid teenuse operatsioonidel. Lisaväärtusena võiks olla ka testandmetega keskkond, kus käivitatud päringud koos avaldistega ei teeks muudatusi kliendi andmetega. Nimetatud liides oleks sarnane lahendus GraphQL *Playground*-ile [50], kus API kasutaja saab koostada dünaamilisi päringuid ja neid käivitada. Nii ei ole kasutajal ka vajadust Visual Studio keskkonnas genereerida JSON kujul avaldisi, sest see sisend tekitatakse liideses.

Kuna andmetega juhitavus antud töös kirjeldatud moel on API kliendile ebamugav ning keeruline protsess, siis ei pruugi kliendid olla huvitatud selle kasutamisest. Nii tekib "surnud kood", mis arendati võimaldamaks andmetega juhitavust, kuid mida reaalselt ei kasutata. Kui graafilise liidese abil teha kliendile avaldiste loomise protsess mugavamaks, siis kasutataks seda võimekust tõenäoliselt rohkem. See õigustaks ka lisapingutusi teenuse arendusprotsessis, mille tulemusena on andmetega juhitavus võimalik.

5 Muuda C# programmeerimiskeeles arendatud REST protokollil põhinev rakendusliidese andmetega juhitavaks

Antud peatükis esitatakse juhend REST protokollil põhineva C# programmeerimiskeeles arendatud rakendusliidese (API) andmetega juhitavaks muutmiseks. Juhendi aluseks on peatükis 4 teostatud tarkvara andmetega juhitavaks muutmise protsess. Tarkvara (sh tarkvara disaini) alaste teadmiste mustrite formaadis struktureeritult väljendamine on levinud lähenemine [51] [52]. Seega pannakse loetavuse ja võrreldavuse huvides juhend kirja ühte võimalikku disainimustri formaati kasutades (võimalikke formaate, mis määravad mustri struktuurielemendid, on erinevaid). Mustrikandidaadi nimi on peatüki pealkiri. Mustrite puhul tuleb järgida kolme korra reeglit [53], mis tähendab, et probleemiahendus on muster, kui selle kasutamise kohta on vähemalt kolm näidet (st selline lahendus pole enam juhus). Kuna siin pakutava lahenduse kasutamise kohta ei ole kolme näidet, siis on tegemist mustri kandidaadiga.

Käesolevas töös loodud lahendus põhineb C# programmeerimiskeelel ja selle *Expression* baasklassi avaldistel. Sarnast lahendust saaks kirjeldatud töövoogude ja avaldiste esitamise loogika alusel kohandada ka teistes programmeerimiskeeltes arendatud teenustele, kuid see nõuab API sisendi lugemisel ja avaldiste esitamisel potentsiaalselt lisapingutusi. Olenevalt programmeerimiskeele võimalustest võib olla vaja luua avaldiste käivitamiseks lisafunktsionaalsus, kui keelel puudub C# *Expression* klassile sarnane avaldiste ja nende käivitamise võimekus. Antud töös neid lahendusi ei käsitleta, st mingite teiste programmeerimiskeelte põhist lahendust ei looda.

5.1 Eesmärk

- Kirjeldada samm-sammuline juhend C# programmeerimiskeeles arendatud REST protokollil põhineva rakendusliidese rakenduse andmetega juhitavaks muutmiseks läbi C# keele avaldistepuu.
- Kirjeldada andmetega juhitava äriprotsessi töövoog GET ja POST tüüpi päringutele.

5.2 Probleem

Mitmevalduslike süsteemide puhul on tarkvara üks olulisemaid omadusi kõrge konfigureeritavuse tase [3]. Süsteemi klientide nõudmised süsteemile ei ole täpselt ühesugused ning süsteemi kohandamine võimalikult suurele osale klientidele sobivaks nõuab teenusepakkuja poolelt suurt pingutust. Tihti tähendab nõude täitmine teenusepakkuja poolse arenduse läbi kliendile pikka ooteaega. Üldine "üks suurus sobib kõigile" süsteem tähendab kliendile detailsete nõuete arvelt suuremaid järeleandmisi kui üksikvalduslik süsteem (süsteem, mida klient ise arendab).

Üldist lahendust pakkuv teenus on rakendusliidese (API) kasutajale konfigureeritav vaid liidesesse sisseehitatud võimekuse ulatuses. Äriloogikat realiseerivad avaldised, mis on koodi tasemel defineeritud ning ei ole sisendandmetega juhitavaks muudetud, ei ole kliendi poolt tegelikult kontrollitavad. Kui kliendil puudub teenuses enda andmete ja nendega teostavate tegevuste üle kontroll, siis võib klient teenusepakkujat vahetada. Ärilises plaanis tähendab see teenusepakkujale rahalist kaotust.

5.3 Kirjeldus

Teenuse kasutaja poolt andmetega juhitav tarkvarateenus (edaspidi andmetega juhitav tarkvara/teenus/süsteem) pakub teenuse kasutajale võimaluse mõjutada tarkvara käitumist sellele mingil viisil etteantavate sisendandmetega. Käitumist juhtivaid sisendandmeid on ühe võimalusena võimalik ette anda teenuse väljakutsel läbi selle sama API, mida klient kasutab teenusepakkuja teenuse tarbimiseks. Juhtimist võimaldavaks API sisendiks võib olla JSON formaadis esitus koodis kompileeritavast ja dünaamiliselt käivitatavast avaldistepuust. Sisendiks olnud avaldis või avaldised käivitatakse kas koos põhifunktsionaalsusega või hoopiski selle asemel. Käesolevas juhendis kirjeldatakse just sellisel viisil andmetega juhitavuse realiseerimist.

Andmetega juhitavus pakub võimaluse lugeda või muuta andmebaasi olemite atribuutide ja seoste väärtuseid dünaamiliselt, vastavalt etteantud juhistele. Mitmevalduslikus jaotusmudelis on selline lahendus kasulik, kui ühe klientsüsteemi nõuded erinevad teistest klientsüsteemidest. Nii on võimalik soovitud muudatus võtta sisendandmete läbi kasutusele vaid konkreetse klientsüsteemi korral.

Andmetega juhitavus antud mõistes on mõistlik lahendus sellistele teenuseoperatsioonidele, kus arvutatakse väärtuseid või tehakse otsuseid kasutaja sisendi või andmebaasis olevate väärtuste põhjal. Selle asemel, et leida arvutatud väärtus koodis kirjeldatud valemi põhjal, on võimalik valem operatsioonile koos teiste sisendandmetega operatsiooni väljakutses ette anda. Hea näitena sellistest süsteemidest võib tuua hindasid või koguseid arvutavad süsteemid, kus valemid võivad olla ajas muutuvad.

Andmetega juhitavus võimaldab kliendile paindlikkust, kuid tähendab nii süsteemi arendamisel kui ka süsteemi liideste integreerimisel suuremat keerukust.

5.4 Struktuur

Andmetega juhitav süsteem on defineeritav läbi kahe peamise omaduse, mille koostoimel on võimalik pidada süsteemi andmetega juhitavaks: avaldiste esitamine API sisendina ja avaldiste väärtuste arvutamine teenuse täitmise käigus.

Andmetega juhitav süsteem peab aktsepteerima oma sisendina lisaks teenuse defineeritud operatsiooni toimimiseks vajalike parameetrite väärtustele ka avaldist või avaldiste massiivi. Nii arendatud API kontrolleri lõpp-punktid sarnanevad oma olemuselt pisut GraphQL [36] [37] skeemi põhiselt defineeritud lõppsõlmele, mis on olenemata päringust POST tüüpi. Selliselt on võimalik HTTP päringu kehas defineerida sisendiks avaldist. Neid päringuid, mis juba võimaldavad kehas andmete edastamist, see omadus eriti ei muuda – lisandub vaid üks sisendparameeter. Joonis 48 on üldine GET päring esitatud tavapärasel ning andmetega juhitaval kujul:

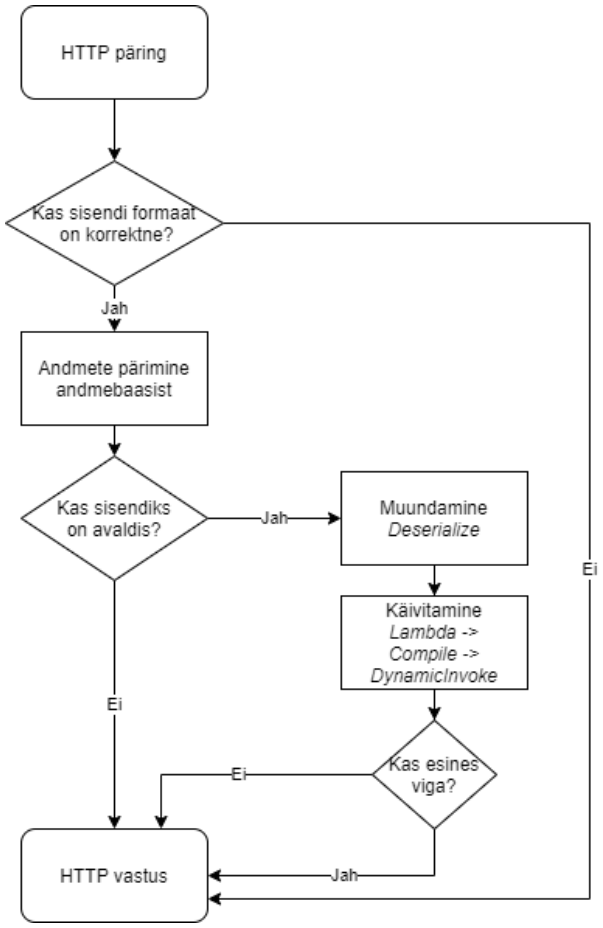
```
[HttpGet("{id}")]
public IActionResult GetEntity([FromRoute] Guid id)
{
      return Ok(_service.Get(id));
}

[HttpPost("{id}")]
public IActionResult GetEntity([FromRoute] Guid id,
[FromBody] EntityExpression expression)
{
      return Ok(_service.Get(id, expression));
}
```

Joonis 48: GET päring tavapärasel ning andmetega juhitaval kujul.

Teine struktuurne muudatus teenuste andmetega juhitavuse võimaldamiseks on äriloogika kihis sisendi muundamine kompileeritavateks avaldisteks. Sisendparameetrina olev JSON vormis avaldis muundatakse *Deserialize<Expression>()* meetodi abil *Expression* tüüpi objektiks, mida on võimalik koodis kompileerida ja käivitada.

Programmi põhifunktsionaalsuse toimimise tagamiseks peab äriloogika kihi meetodites järgima kindlat töövoogu. Nii on võimalik kindlustada teenuse kliendile, et kui sisendis pole avaldisi, siis töötab põhifunktsionaalsus. Samuti tagatakse nii põhifunktsionaalsuse toimimine, juhul kui avaldise käivitamisel peaks ette tulema viga.

Andmete teenuse kaudu andmebaasist pärimisel (otsimisel) on töövoog päringu saatmise algusest päringu vastuse saamiseni süsteemis esitatud Joonis 49. Käivitatav sündmus ja lõppseisund esitatakse ümarate servadega ristkülikuna, toimingud teravate servadega ristkülikuna ning otsustuspunkt rombina.

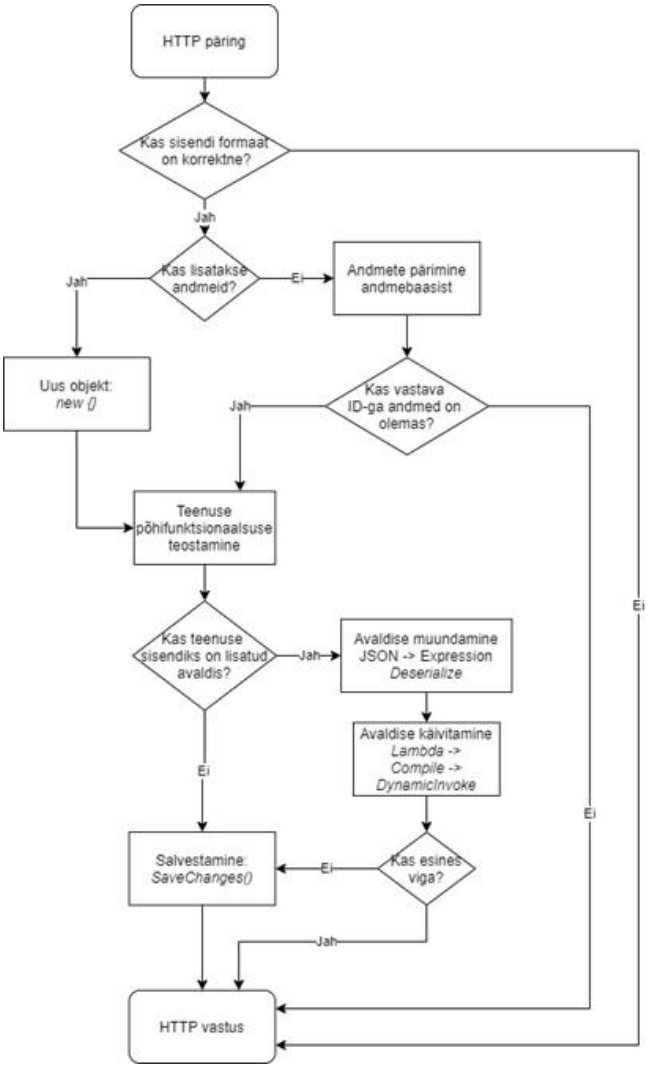


Joonis 49: Andmete otsimise töövoog.

Joonis 49 oleva vooskeemi kohaselt teostatakse avaldise muundamine ja käivitamine põhivoo kõrval vaid juhul, kui avaldis on esitatud päringu sisendis. Muul juhul käivitatakse vaid põhifunktsionaalsus.

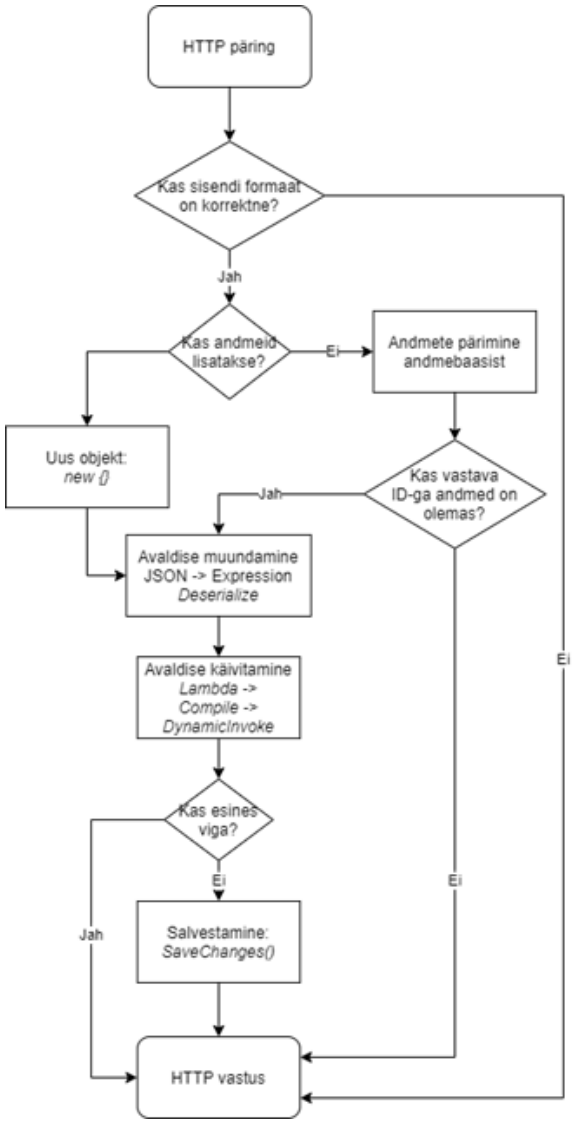
Andmemuudatusi teostavates protsessides on eristatavad kaks erinevat töövoogu: täielik ja täiendav andmetega juhtimine.

Täiendava andmetega juhtimise korral käivitatakse programmi põhifunktsionaalsus ja selle järel etteantud avaldis nagu kirjeldab Joonis 50.



Joonis 50: Täiendavalt andmetega juhitava rakenduse töövoog.

Täieliku andmetega juhtimise korral käivitatakse ainult etteantud avaldis. Põhifunktsionaalsusest teostatakse ainult andmete lisamine või muutmine – muid äriloogilisi tegevusi ei tehta.



Joonis 51: Täielikult andmetega juhitava rakenduse töövoog.

Mõlema kirjeldatud töövoo puhul on sarnane salvestamise loogika. Andmebaasi salvestatakse andmed pärast kogu programmi töövoo läbimist. Kui programmi põhifunktsionaalsuses või käivitatud avaldises esineb viga, siis andmeid ei salvestata ja tagastatakse vastav HTTP vastus. See on oluline, et andmebaasi ei salvestataks poolikuid ja vigaseid andmeid.

5.5 Näide

Andmetega juhitavuse eesmärk on pakkuda operatsiooni väljakutse sisendina võimalust defineerida avaldisi, mis teevad arvutusi andmebaasist päringuga loetud andmete põhjal ning esitavad tulemuse päringu vastuses või arvutavad väärtuseid enne kui need andmebaasi salvestatakse. Selle näiteks võib olla toote kohta käiva rea lisamine

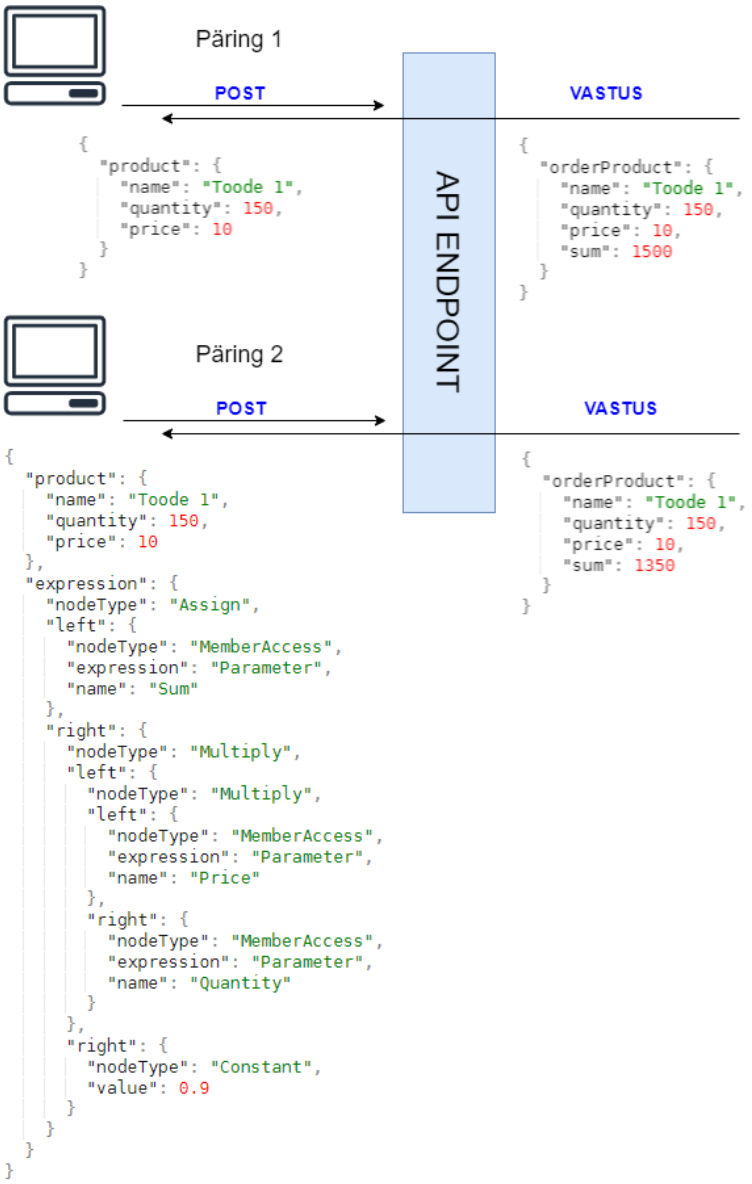
(*AddOrderProduct*) mingisuguse tellimuse või jaotuskava külge. Oletame, et toote ja tellimuse sidumiseks luuakse tellimuse rida Joonis 52 esitatud kujul.

Joonis 52: Tellitava toote andmemudel.

Näite puhul on tellimusse valitud toote summa (*Sum*) väärtus, mis arvutatakse toote tellimusega sidumisel (tellimuse rea loomisel) kasutades valemit *Quantity * Price*.

Andmetega juhitava teenuse põhiomadus tuleb välja siis, kui ühel või mitmel, aga mitte kõikidel teenuse klientidel, on vaja realiseerida rakenduse põhifunktsionaalsust asendav või täiendav tingimus. Näiteks on mingi kindla toote lisamisel vaja realiseerida tingimus, et kui lisatud toote kogus on suurem kui 100 ühikut, siis on hind 10% soodsam.

Joonis 53 kujutab visuaalselt kahte erinevat operatsiooni väljakutset. Joonisel esitatud "Päring 1" on andmete sisestamise operatsiooni väljakutse, mille tulemusel salvestatakse ja tagastatakse vastusena tellimuse rida "*OrderProduct*". Tellimusse valitud toodete summa arvutamise protsess on rakenduse koodis realiseeritud valemiga. "Päring 2" saab sisendiks (joonisel esitatud lihtsustatud kujul JSON formaadis) avaldise. Antud avaldise käivitamisel arvutatakse toodete summa valemiga 0.9 * (toote_hind * toodete_arv). Vastustest on näha, et teise operatsiooni väljakutse põhjal arvutatakse toodete koguhinnaks 1500 asemel 1350, mis on oodatud tulemus.



Joonis 53: Andmetega juhitava päringu võrdlus tavaprotsessiga.

Käesolevas töös kirjeldatakse andmetega juhtimist jaotustarkvara tootekoguste regulatsiooniprotsessi näitel (peatükk 4). Käesolevas jaotises kirjeldatakse lisaks toote hinnaarvutust andmetega juhitava protsessina. Lisaks nimetatud näidetele on võimalik andmetega juhitavaid protsesse leida ka teistest valdkondadest, kus ei ole tegu otseselt toodete müügi ja kogustega. Antud töö autori bakalaureusetöös [54] analüüsiti ja arendati elektroonilise päeviku teenust. Laiendades päevikutarkvara nõudeid, on võimalik ka sealt leida nõudeid, mille täitmiseks võiks tarkvara olla andmetega juhitav.

1. Töö hinde arvutamine – Eeldusel, et kool või muu teenust kasutav asutus annab tööde hindamisel punkte, on võimalik defineerida hinde arvutamiseks muutuvaid valemeid. Näiteks kui tavaliselt on 90% tulemuse

korral hindeks „5“ või „A“, siis oleks andmetega juhitava teenuse puhul defineerida avaldised, kus tunnikontrolli puhul oleks hinde arvutamisel hinde „suurepärane“ saamise lävendiks 95% ja kontrolltöö puhul jääks see lävend samaks.

2. Koondhinde arvutamine – Sarnaselt eelmisele näitele on võimalik hinnatud tööde tüüpide järgi defineerida töö tulemuse tähtsust koondtulemuste arvutamisel. Näiteks koondhinde väärtuse arvutamisel on tunnikontrollide osakaal koguhindest 30%, kontrolltööde osakaal 70%.
3. Tööde kohustuslikkus – Tavaline on, et mingite heade tulemuste saavutamine varasemates töödes tähendab mõnest edasisest kontrolltööst või tunnikontrollist vabastust. Selliselt oleks võimalik andmetega juhtimise teel kontrollida, et kui isikul on varasemate tööde tulemused või koondtulemus näiteks üle 90%, siis muudetakse mingi järgnev töö isikule mittekohustuslikuks.
4. Automaatne negatiivne tulemus – Sarnaselt regulatsioonile jaotustarkvaras on võimalik päevikutarkvaras defineerida „regulatsioon“, et kui isikul on mingi hulk varasemaid töid tegemata või sooritatud mitterahuldavale tulemusele, siis määratakse koondtulemuseks automaatselt „mitterahuldav“.

5.6 Tingimused

API andmetega juhitavaks muutmisel on eelnevalt oluline lähtuda süsteemi eripäradest ning sellest tulenevatest tingimustest.

1. Tuleb veenduda, et rakenduse andmetega juhitavaks muutmine ei põhjustaks rakenduse põhifunktsionaalsuse muutumist. Kui teenus ja selle klientide vajadused eeldavad nii täielikku kui ka täiendavat andmetega juhtimist, siis tuleb vastavalt pakkuda nii põhifunktsionaalsust kui ka andmetega juhitavat funktsionaalsust esitavad API lõppsõlmed. Selliselt on võimalik tagada teenuse toimimine nendele klientsüsteemidele, kes vajavad vaid API põhifunktsionaalsust.
2. Tuleb tagada andmete eraldatus – ühe klientsüsteemi päring ei tohi omada ligipääsu teise klientsüsteemi andmetele. Kirjeldatud viisil andmetega juhtimise eelduseks on ühisavalduslikus süsteemis andmete eraldamine valdajapõhistesse andmebaasidesse. Iga valdaja, kes API päringuid teostab, saab programmikoodi tasemel ligipääsu vaid

enda andmebaasile. Kui valdajatel oleks jagatud andmebaas, siis ühiskasutuses olevate tabelite, näiteks klassifikaatorid, seoste kaudu oleks võimalik avaldistes ka pärida teiste valdajate andmeid. Avaldiste sisus ei ole võimalik programmikoodis käivitamise hetkel enam defineerida kitsendusi, milliseid kirjeid tohib vastav valdaja lugeda. Alternatiivina pakuks kaitset ka ühises andmebaasis andmebaasisüsteemi poolt reapõhiste turvapoliitikate rakendamine, mis kirjutavad ette tingimused, millistele vastavaid ridu saab kasutaja kasutada ja millest ei ole võimalik mööda minne.

3. Klientidele on vaja teha testprojekti avaldiste loomiseks ja testimiseks. Avaldise koostamiseks on eelistatud graafiline liides – sellel peaks olema soovituste süsteem, mis avaldise koostamisel pakub andmemudeli põhjal soovitusi. Sellise liidesena on võimalik kasutada Visual Studio programmeerimiskeskkonda, mis võimaldab kirjutada avaldiste programmikoodi ning muundada selle JSON formaati. Visual Studio soovitustesüsteemi võimekuse täielikuks kasutamiseks tuleks testprojekti jaoks muuta avalikuks andmete ligipääsukiht (antud töös klassiteek *RegulationApi.DAL* [39]), selles defineeritud objekt-relatsioonilised klassid ning nende seosed.
4. Avaldiste käivitamise töövoog tuleb teenust arendades programmikoodis realiseerida korrektselt, et tagada andmete terviklikkus ja korrektsus ka vigaste avaldiste korral. Andmete salvestamisega andmebaasis lõppevate operatsioonide korral on oluline, et vigaseid andmeid ei salvestataks andmebaasi. Kui rakenduses esineb avaldise käivitamisel viga, siis ei tohi andmebaasis andmeid salvestada, vaid tuleb HTTP vastuses tagastada veateade.
5. Kliendile tuleb pakkuda võimalust kasutada teenust ilma avaldisteta – st põhifunktsionaalsuse kasutamisel ei tohi liidese integreerimine ja päringute saatmine nõuda kliendipoolseid lisapingutusi. Üksiku vastutuse ja otstarbe lahususe printsiipide [42] seisukohalt oleks selleks erinevad API lõppsõlmed eelistatud. Põhifunktsionaalsust realiseeriv API lõppsõlm oleks andmetega juhitavate sõlmede aluseks, kuid oleks eraldiseisev ning ei lubaks sisendina avaldisi.

5.7 Tähelepanekud

Teenuse andmetega juhitavaks muutmine nõuab lisapingutusi nii arendaja kui ka teenuse kasutaja poolelt. Käesolevas jaotises tuuakse välja mõned punktid, millele tuleb andmetega juhitava teenuse loomisel mõelda.

- Kirjeldatud kujul andmetega juhitav süsteem on ilma avaldiste koostamiseks mõeldud graafilise liideseta toimiv, kuid kasutamiseks keeruline lahendus. Visual Studio keskkond võimaldab konsoolirakenduse abil avaldiste koostamist ja nende muundamist JSON formaadis avaldiseks. Ilma spetsiaalse graafilise liideseta on Visual Studio parim lahendus keerukamate avaldiste koostamiseks. Samas eeldab Visual Studios avaldiste koostamine, et arendajal on teadmine teenuse andmemudelist ja klasside omavahelistest seostest. Graafilise liidese olemasolu võimaldaks luua avaldisi soovituste alusel, kuid liidest integreeriva süsteemi arendajal ei ole sellisel juhul vajadust rakenduse andmemudelit tunda.
- Andmetega juhitavus on testitav ja realiseeritav ka süsteemi osaliselt muutes. Teenuse või tarkvara andmetega juhitavaks muutmiseks vajaliku lisapingutuse ja sellest tuleneva kasu hindamiseks on võimalik muuta vaid üht teenust või süsteemi osa. Erinevate tingimuste ja andmetega testimine võimaldab valideerida, kas tehtud lisapingutused on arendatava süsteemi mõistes õigustatud.
- Andmetega juhitav tarkvara võib tingida süsteemile suurenenud koormuse. Sisendiks oleva avaldise arvutamine võtab programmikoodis vajaliku teisenduse ja käivitamise tõttu rohkem aega kui sama funktsionaalsuse realiseerimine otse programmikoodis. Lisaks on avaldiste puhul võimalus käivitada keerukamaid päringuid või korduseid, mis veelgi pikendaksid vastamiseks kuluvat aega. Selleks, et vältida süsteemi ülekoormust, tuleks seada etteantud avaldiste programmikoodis käivitamiseks ajalised piirangud. Kui süsteemi kliendi avaldised jäävad piirangute tõttu käivitamata, siis on teenuse arendajal võimalik kontrollida sisendiks olnud avaldist ning vajadusel soovitada kliendile optimaalsemat lahendust.
- Teenuse valdajad peavad tagama, et nende süsteemist saadetud päringud ja avaldised ei oleks omakorda selle klientsüsteemide kasutajate poolt kontrollitavad. Kui teenuse integreerinud klientsüsteemi lõppkasutajatel on ligipääs avaldistele, mis teiste osapoolte poolt teenusele saadeti, siis võidakse seda avaldist muuta endale kasulikus suunas (ning teistele klientidele ja

teenusepakkujale kahjulikus suunas). Andmetega juhitava teenuse jaoks on valdaja poolt saadetud avaldised käsuks ning teenus eeldab, et avaldise on koostanud vastava süsteemi arendaja või administraator.

6 Kokkuvõte

Käesoleva töö üheks eesmärgiks oli kavandada ja realiseerida REST protokollil põhinev andmetega juhitav teenus. Töö käigus arendatud näidisprojekti põhjal oli teiseks eesmärgiks koostada disainimustri formaati kasutades juhend andmetega juhitavate C# keeles kirjutatud REST liideste arendamiseks.

Tarkvara andmetega juhitavus lõputöö probleemipüstituses kirjeldatud viisil ei ole väga levinud praktika. Tausta uurimise käigus selgus, et ülesehituse ja üldise töövoo mõistes on üks lähemaid alternatiive GraphQL [33]. GraphQL on valmiduse tasemelt terviklik ja suure kasutajate hulgaga päringukeel. Samas ei võimalda antud lahendus probleemipüstituses kirjeldatud viisil sisendandmete kaudu manipuleerida tarkvara äriloogika kihis olevate operatsioonidega.

Töö praktilises osas loodi jaotustarkvara regulatsioonidega seotud osasüsteem, mis antud töö kontekstis on näidisprojektiks. Selle tarbeks kirjeldati töö mõistes olulised kasutusjuhud ning loodi andmebaasi disaini kirjeldav diagramm. Näidisprojekti nõuded kirjeldati töö aluseks olevas ajaveebi postituses [11] tutvustatud regulatsiooniprotsessi loogika põhjal. Näidisprojekti esimeses iteratsioonis arendati regulatsiooniprotsessi põhifunktsionaalsus REST API liidesena.

Projekti teises etapis täiendati esimeses etapis loodud tarkvara. Muutunud äriloogika nõude realiseerimiseks muudeti olemasolev rakendus ja selle funktsionaalsus andmetega juhitavaks. Pakutud lahenduseks on C# avaldistel ja avaldistepuu struktuuril põhinev loogika. Avaldistepuu antakse API sisendisse JSON formaadis ning päringu käivitamise hetkel muundab, kompileerib ja käivitab äriloogika kihi teenus avaldise, andes sellele argumendiks andmebaasist küsitud andmed. Lahenduse GET ja POST tüüpi päringutele defineeriti selles etapis äriloogikakihi töövood.

Teises etapis kirjeldatud uue nõude ja tehtud muudatuste põhjal saab öelda, et pakutud lahendus rahuldas ärilisi vajadusi. Lisaks võimaldab see tulevikus sarnaseid muudatusi lihtsamini teostada. Sellist lahendust ilmestab kõrge konfigureeritavuse tase – klient ei sõltu nii suurel määral teenusepakkuja poolsetest arendustest. Teisalt on antud kujul lahendust raske kasutada – API sisendi ülesehitus on keeruline ning keerulisemate avaldiste kirjeldamine on aeganõudev protsess.

Töös arendatud näidisprojekti põhjal kirjeldati töö viimases etapis disainimustri formaati järgides andmetega juhitava REST protokollil põhineva C# keeles kirjutatud rakendusliidese saavutamise juhend. Kuna sellise lahenduse kasutamise kohta ei ole veel kolme näidet, siis on tegemist mustri kandidaadiga. Juhendi aluseks on pakutud C# avaldistel põhinev lahendus. Juhendis kirjeldatakse sellisel lahendusel põhineva teenuse struktuur ning töövood andmete otsimiseks ja muutmiseks koos näitega.

Nii lõputöö praktilise osa hindamise kui ka disainimustri formaadis juhendi tähelepanekute all on välja toodud pakutava lahenduse puudujäägid. Kuigi antud projekti kontekstis on pakutav lahendus funktsionaalselt töötav, ei ole see eraldiseisvana püstitatud probleemile mugav lahendus. Sisendiks olevate avaldiste JSON kujule viimine nõuab teenuse andmebaasiskeemi tundmist ja Visual Studio keskkonna kasutamist. Võimaliku edasiarendusena antud tööle võiks kaaluda graafilise liidese loomist. Selline graafiline keskkond võiks töötada niinimetatud “liivakastina”, kus oleks võimalik avaldisi koostada ja teisendada API sisendiks sobivale kujule. Lisaks peab sellist süsteemi arendades silmas pidama, et teenuse koormus suureneb olenevalt sisendiks olevate avaldiste keerukusest. Töös tehti kahele REST API-le koormustestimine, milledest üks oli andmetega juhitav ja teine mitte. Koormustestide tulemustest selgus, et koormuse suurendamisel pikenes andmetega juhitava API vastamisaeg ja suurenes päringute hulk, mis said ajalimiidi ületamise veateate. Kuna sisendi muundamine ja käivitamine on ajakulukam kui sama funktsionaalsuse koodina esitamine, peab andmetega juhitava API korral arvestama selle mõjuga jõudlusele. Samuti on teenuse klientide andmete turvalisuse tagamiseks oluline, et klientide andmed oleks eraldatud. Üheks võimaluseks on, et iga teenuse valdaja andmed peaksid olema ainult temale mõeldud andmebaasis ning teenuse päringu käigus on ligipääs vaid selles baasis olevatele andmetele. Teine võimalus on, et valdajate ühise andmebaasi korral tagatakse andmebaasisüsteemi rakendatavate reapõhiste poliitikatega, et teenuse valdajal on tabelis juurdepääs ainult endaga seotud ridadele.

Autor leiab, et töös püstitatud eesmärgid saavutati. Disainimustrite formaadis valminud juhendi valideerimiseks on võimalik kirjeldatud sammudega teha läbi uus arendusprotsess. Töö praktilises osas valminud tarkvara on võimalik selles protsessis käsitleda näidisena. Antud töö tulemusel kirja pandud disainimustri formaadis juhend on disainimustri kandidaat.

Kasutatud kirjandus

[1] S. Satyanarayana, „Cloud computing: SAAS,“ *Computer Sciences and Telecommunications,* nr 4, pp. 76-79, 2012.

[2] R. Krebs, C. Momm ja S. Kounev, „Architectural Concerns in Multi-tenant SaaS Applications.,“ *Closer,* nr 12, pp. 426-431, 2012.

[3] C.-P. Bezemer ja A. Zaidman, „Multi-tenant SaaS applications: maintenance dream or nightmare?,“ *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE),* pp. 88-92, 2010.

[4] R. C. Martin, Clean code: a handbook of agile software craftsmanship, Upper Saddle River, NJ: Prentice Hall, 2008.

[5] „Source Making,“ [Võrgumaterjal]. Available: https://sourcemaking.com/refactoring/smells.

[6] M. Turner, D. Budgen ja P. Brereton, „Turning software into a service,“ *Computer,* nr 10, pp. 38-44, 2003.

[7] M. M. Lehman, „Laws of software evolution revisited.,“ *European Workshop on Software Process Technology.,* pp. 108-127, 1996.

[8] M. Kim, N. Meng ja T. Zhang, Software Evolution - Handbook of Software Engineering, Cham: Springer, 2019, pp. 223-284.

[9] S. Sidiroglou, S. Ioannidis ja A. D. Keromytis, „Band-aid patching,“ *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability,* 2007.

[10] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy ja L. Bairavasundaram, „How do fixes become bugs?,“ *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering,* pp. 22-36, september 2011.

[11] K. Downs, „Minimize Code, Maximize Data,“ 4 mai 2008. [Võrgumaterjal]. Available: http://database-programmer.blogspot.com/2008/05/minimize-code-maximize-data.html. [Kasutatud 11 mai 2020].

[12] „Language Integrated Query (LINQ),“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/. [Kasutatud 11 mai 2020].

[13] „Expressions (C# Programming Guide),“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expressions. [Kasutatud 11 mai 2020].

[14] V. Vaishnavi, B. Kuechler ja S. Petter, „Design Science Research in Information Systems,“ 2004. [Võrgumaterjal]. Available: http://www.desrist.org/design-research-in-information-systems/. [Kasutatud 11 mai 2020].

[15] A. Hevner ja S. Chatterjee, „Design science research in information systems,“ *Design research in information systems,* pp. 9-22, 2010.

[16] K. Wondra, „End user development - An asset or a liability?,“ 13 november 2012. [Võrgumaterjal]. Available: https://www.skylinetechnologies.com/Blog/Skyline-Blog/November-2012/End-User-Development-An-Asset-or-a-Liability. [Kasutatud 11 mai 2020].

[17] „Star business solutions. Is a SaaS business software solution actually more expensive?,“ [Võrgumaterjal]. Available: https://www.starbusinesssolutions.com.au/insights/is-saas-more-expensive. [Kasutatud 11 mai 2020].

[18] M. Chisholm, How to build a business rules engine: extending application functionality through metadata engineering, San Francisco: Morgan Kaufmann, 2004.

[19] „Business rules engine,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Business_rules_engine. [Kasutatud 19 mai 2020].

[20] „business rules engine (BRE),“ [Võrgumaterjal]. Available: https://searchapparchitecture.techtarget.com/definition/business-rules-engine-BRE. [Kasutatud 20 mai 2020].

[21] J. Voronova, „PostgreSQL SQL lausete paindlikuks muutmiseks mõeldud infosüsteemi kavandamine,“ TTÜ, Tallinn, 2014.

[22] „APEX,“ [Võrgumaterjal]. Available: https://apex.oracle.com/en/. [Kasutatud 15 mai 2020].

[23] R. Raidma, „PostgreSQL andmebaasisüsteemi põhine metaandmetega juhitavate veebirakenduste kiirprogrammeerimise keskkond,“ TTÜ, Tallinn, 2016.

[24] N. Kopa, „PostgreSQL andmebaassüsteemi põhise metaandmetega juhitavate veebirakenduste kiirprogrammeerimise keskkonna edasiarendus,“ TTÜ, Tallinn, 2019.

[25] H. Hai ja S. Sakoda, „SaaS and integration best practices,“ *Fujitsu Scientific and Technical Journal,* kd. 45, nr 3, pp. 257-264, 2009.

[26] „Expression Trees (C#),“ Microsoft, 2015. [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/. [Kasutatud 11 mai 2020].

[27] „Expression Class,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.expression?view=netcore-3.1. [Kasutatud 11 mai 2020].

[28] M. Konicek, „How to implement a rule engine in C#,“ [Võrgumaterjal]. Available: http://coding-time.blogspot.com/2011/07/how-to-implement-rule-engine-in-c.html. [Kasutatud 11 mai 2020].

[29] „Work with Language-Integrated Query (LINQ),“ 2018. [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/working-with-linq#introduction. [Kasutatud 11 mai 2020].

[30] „End-user development,“ [Võrgumaterjal]. Available: https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/End-user_development.html. [Kasutatud 11 mai 2020].

[31] R. Kuusik, „Infosüsteemi isearendamise toimimismudel infoühiskonnas,“ *A & A,* nr 3, p. 14/22, 2000.

[32] „Low-code development platform,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Low-code_development_platform. [Kasutatud 11 mai 2020].

[33] „GraphQL,“ [Võrgumaterjal]. Available: https://graphql.org/. [Kasutatud 11 mai 2020].

[34] S. Greif, „So what’s this GraphQL thing I keep hearing about?,“ 11 aprill 2017. [Võrgumaterjal]. Available: https://www.freecodecamp.org/news/so-whats-this-graphql-thing-i-keep-hearing-about-baf4d36c20cf/. [Kasutatud 11 mai 2020].

[35] „Queries and Mutations,“ [Võrgumaterjal]. Available: https://graphql.org/learn/queries/. [Kasutatud 11 mai 2020].

[36] „EntityGraphQL,“ [Võrgumaterjal]. Available: https://github.com/lukemurray/EntityGraphQL. [Kasutatud 11 mai 2020].

[37] „Code C#/.NET,“ [Võrgumaterjal]. Available: https://graphql.org/code/#c-net. [Kasutatud 11 mai 2020].

[38] „Entity Framework documentation,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/ef/. [Kasutatud 11 mai 2020].

[39] S. Puntso, „Andmetega juhitava API programmikoodi repositoorium,“ [Võrgumaterjal]. Available: https://gitlab.com/sanderpuntso/data-driven-api.

[40] „How to use expression trees to build dynamic queries (C#),“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/how-to-use-expression-trees-to-build-dynamic-queries. [Kasutatud 11 mai 2020].

[41] „ExpressionJsonSerializer,“ [Võrgumaterjal]. Available: https://github.com/aquilae/expression-json-serializer. [Kasutatud 11 mai 2020].

[42] „Single-responsibility principle,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Single-responsibility_principle. [Kasutatud 15 mai 2020].

[43] „Security Considerations (Entity Framework),“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/cc716760(v=vs.100)?redirectedfrom=MSDN. [Kasutatud 15 mai 2020].

[44] „Row-Level Security,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver15. [Kasutatud 19 mai 2020].

[45] „Row level security in EntityFramework 6 (EF6),“ [Võrgumaterjal]. Available: https://docs.microsoft.com/et-ee/archive/blogs/mvpawardprogram/row-level-security-in-entityframework-6-ef6. [Kasutatud 15 mai 2020].

[46] „Kerckhoffs's principle,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Kerckhoffs%27s_principle. [Kasutatud 15 mai 2020].

[47] „Apache JMeter,“ [Võrgumaterjal]. Available: https://jmeter.apache.org/. [Kasutatud 15 mai 2020].

[48] „Load Testing Tutorial: What is? How to? (with Examples),“ [Võrgumaterjal]. Available: https://www.guru99.com/load-testing-tutorial.html. [Kasutatud 15 mai 2020].

[49] „Dead Code,“ [Võrgumaterjal]. Available: https://sourcemaking.com/refactoring/smells/dead-code. [Kasutatud 11 mai 2020].

[50] „GraphQL Playground,“ [Võrgumaterjal]. Available: https://www.apollographql.com/docs/apollo-server/testing/graphql-playground/. [Kasutatud 11 mai 2020].

[51] „Design Patterns,“ [Võrgumaterjal]. Available: https://sourcemaking.com/design_patterns. [Kasutatud 15 mai 2020].

[52] „Design Patterns,“ [Võrgumaterjal]. Available: https://refactoring.guru/design-patterns. [Kasutatud 15 mai 2020].

[53] „The Software Patterns Criteria,“ [Võrgumaterjal]. Available: http://antipatterns.com/whatisapattern/. [Kasutatud 15 mai 2020].

[54] S. Puntso, „Elektroonilise huvikoolide päeviku veebiteenuse analüüs ja prototüüp,“ TTÜ, Tallinn, 2017.

Lisa 1 – C# avaldise esitus JSON formaadis

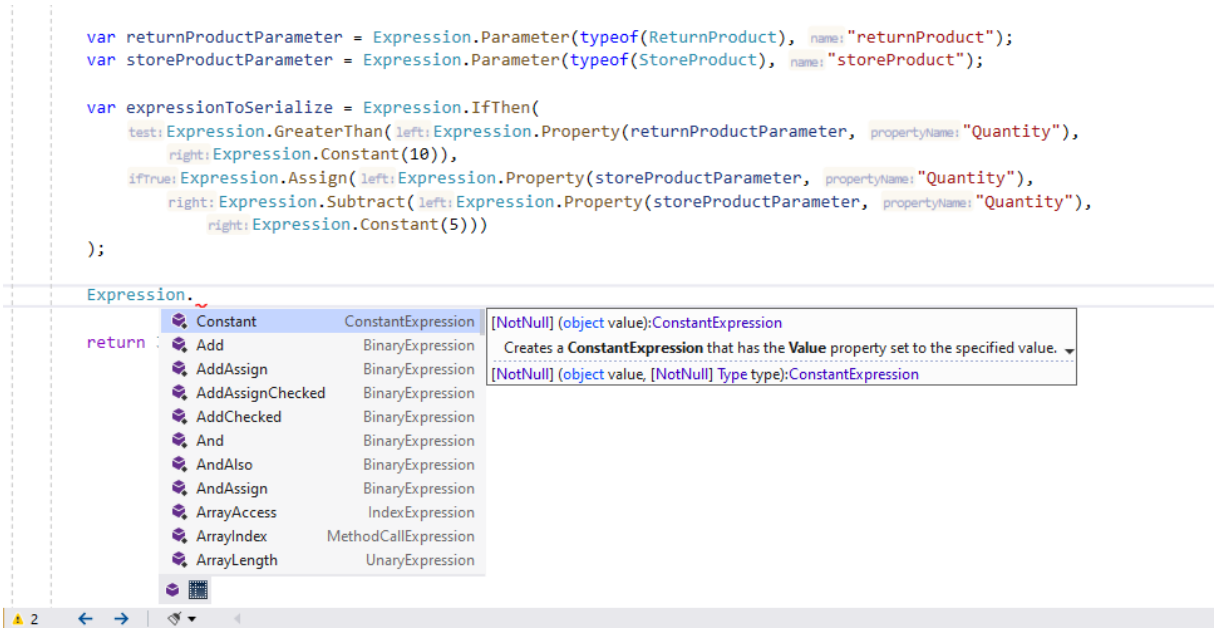
```
{
  "nodeType": "Conditional",
  "type": {
    "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
    "typeName": "System.Void",
    "genericArguments": null
  },
  "typeName": "conditional",
  "test": {
    "nodeType": "GreaterThan",
    "type": {
      "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
      "typeName": "System.Boolean",
      "genericArguments": null
    },
    "typeName": "binary",
    "left": {
      "nodeType": "MemberAccess",
      "type": {
        "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
        "typeName": "System.Int32",
        "genericArguments": null
      },
      "typeName": "member",
      "expression": {
        "nodeType": "Parameter",
        "type": {
          "assemblyName": "RegulationApi.DAL, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null",
          "typeName": "RegulationApi.DAL.Model.ReturnProduct",
          "genericArguments": null
        },
        "typeName": "parameter",
        "name": "returnProduct"
      },
      "member": {
        "type": {
          "assemblyName": "RegulationApi.DAL, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null",
          "typeName": "RegulationApi.DAL.Model.ReturnProduct",
          "genericArguments": null
        },
        "memberType": 16,
        "name": "Quantity",
        "signature": "Int32 Quantity"
      }
    },
    "right": {
      "nodeType": "Constant",
      "type": {
        "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
        "typeName": "System.Int32",
        "genericArguments": null
      },
```

```
      "typeName": "constant",
      "value": {
        "type": {
          "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
          "typeName": "System.Int32",
          "genericArguments": null
        },
        "value": 10
      }
    },
    "method": null,
    "conversion": null,
    "liftToNull": false
  },
  "ifTrue": {
    "nodeType": "Assign",
    "type": {
      "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
      "typeName": "System.Int32",
      "genericArguments": null
    },
    "typeName": "binary",
    "left": {
      "nodeType": "MemberAccess",
      "type": {
        "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
        "typeName": "System.Int32",
        "genericArguments": null
      },
      "typeName": "member",
      "expression": {
        "nodeType": "Parameter",
        "type": {
          "assemblyName": "RegulationApi.DAL, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null",
          "typeName": "RegulationApi.DAL.Model.StoreProduct",
          "genericArguments": null
        },
        "typeName": "parameter",
        "name": "storeProduct"
      },
      "member": {
        "type": {
          "assemblyName": "RegulationApi.DAL, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null",
          "typeName": "RegulationApi.DAL.Model.StoreProduct",
          "genericArguments": null
        },
        "memberType": 16,
        "name": "Quantity",
        "signature": "Int32 Quantity"
      }
    },
    "right": {
      "nodeType": "Subtract",
```

```
      "type": {
        "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
        "typeName": "System.Int32",
        "genericArguments": null
      },
      "typeName": "binary",
      "left": {
        "nodeType": "MemberAccess",
        "type": {
          "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
          "typeName": "System.Int32",
          "genericArguments": null
        },
        "typeName": "member",
        "expression": {
          "nodeType": "Parameter",
          "type": {
            "assemblyName": "RegulationApi.DAL, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null",
            "typeName": "RegulationApi.DAL.Model.StoreProduct",
            "genericArguments": null
          },
          "typeName": "parameter",
          "name": "storeProduct"
        },
        "member": {
          "type": {
            "assemblyName": "RegulationApi.DAL, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null",
            "typeName": "RegulationApi.DAL.Model.StoreProduct",
            "genericArguments": null
          },
          "memberType": 16,
          "name": "Quantity",
          "signature": "Int32 Quantity"
        }
      },
      "right": {
        "nodeType": "Constant",
        "type": {
          "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
          "typeName": "System.Int32",
          "genericArguments": null
        },
        "typeName": "constant",
        "value": {
          "type": {
            "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
            "typeName": "System.Int32",
            "genericArguments": null
          },
          "value": 5
        }
      },
```

```
      "method": null,
      "conversion": null,
      "liftToNull": false
    },
    "method": null,
    "conversion": null,
    "liftToNull": false
  },
  "ifFalse": {
    "nodeType": "Default",
    "type": {
      "assemblyName": "System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e",
      "typeName": "System.Void",
      "genericArguments": null
    },
    "typeName": "default"
  }
}
```

Lisa 2: Avaldise kirjutamine Visual Studio keskkonnas



Lisa 3: Avaldise kirjutamise soovituste pakkumine Visual Studio keskkonnas