TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Karel Vuks 186077IADB

Tarkvaraarendusettevõtte PHP-põhise pärandsüsteemi üleviimine .NET tehnoloogiatele

Bakalaureusetöö

Juhendaja: German Mumma
MSc

Tallinn 2025

Autorideklaratsioon

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

Autor: Karel Vuks

20.04.2025

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on uuendada tarkvaraarendusettevõtte pärandsüsteemi PHP-põhiselt platvormilt .NET põhistele tehnoloogiatele. Tegemist on pärandsüsteemiga millel on hakanud esinema jõudlus ja stabiilsus probleeme , mis häirivad kasutajate igapäevast töövoogu.

Lõputöö käigus analüüsiti pärandsüsteemi kitsaskohti, migreerimisstrateegiaid ja arhitektuurilisi valikuid. Parimaks lähenemisviisiks valiti teenuste järkjärguline üleviimine uuele platvormile, jagades pärandsüsteemis paiknevad teenused funktsionaalselt väiksemateks ja iseseisvateks tagarakendusteks. Analüüsile tuginedes teostatid .NET tehnoloogiatel põhinev tagarakenduse ja Vuel põhinev esirakenduse alus, mille tulemusena on uus süsteem stabiilsem ja jõudlusprobleemideta. Samuti on uus süsteem integreeritud ettevõtte igapäevasesse töövoogu, kus see asendab pärandsüsteemis paiknevaid ärikriitilisi tööriistu.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 31 leheküljel, 8 peatükki, 15 joonist, 1 tabelit.

Abstract

Transitioning Software Development Company's PHP-based Legacy System to .NET Technologies

The aim of this thesis is to update a software development company’s legacy system from a PHP-based platform to. NET-based technologies. The legacy system has started to experience performance and stability issues that disrupt users' daily workflow.

Within the thesis, the author analysed different legacy system bottlenecks, migration strategies and architectural choices. The best approach was to transfer services to the new platform gradually. Where legacy systems' backend services were segmented into smaller independent services. Based on the analysis, .NET technologies were used to develop a backend and Vue was used to create a frontend application. As a result, the newly developed system was more stable and does not have any performance problems. In addition to that, the new system has been integrated into the company’s daily workflow, where it replaces business critical tools from legacy systems.

The thesis is in Estonian and contains 31 pages of text, 8 chapters, 15 figures, 1 tables.

Lühendite ja mõistete sõnastik

.NET	Microsofti poolt loodud raamistik
AMQP	*Advanced Message Queuing Protocol*, sõnumite põhine asünkroonne protokoll mida kasutatakse tarkvarakomponentide vahel
API	*Application Programming Interface*, rakendusliides
BLL	*Business Logic Layer*, äriloogikakiht sisaldab rakenduse äriloogikat ja vahendab andmeid DAL kihist PL kihti
Blocking I/O	Olukord kus klient esitab serverile ühenduse loomise taotluse siis blokeeritakse ühendust haldav lõim seniks kuni on lõpetatud andmete lugemine või kirjutamine
Cache	Vahemälu
Cache-aside	Vahemällu salvestamise strateegia, kus rakendus salvestab vahemällu puuduvad andmeid
CSS	*Cascading Style Sheets*, keel veebilehe kujundamiseks
DAL	*Data Access Layer*, andmekiht sisaldab andmeallikaga suhtlemise loogikat
DLL	*Dynamic-link library*, Windowsi operatsioonisüsteemis kasutatav failivorming
Docker Swarm	Docker konteinerite haldamise tööriist
DTO	*Data Transfer Object*, kihtide vahel asetsev andmeedastusobjekt
HTTP(s)	*Hypertext Transfer Protocol (Secure)*, andmete edastamise protokoll veebis
JS	JavaScript, programmeerimise keel veebirakenduste arendamiseks
JSON	*JavaScript Object Notation*, andmevahetusvorming
JVM	*Java Virtual Machine*, tegemist on virtuaalmasinaga, mis võimaldab operatsioonisüsteemidel jooksutada Java põhiseid rakendusi
LTS	*Long-term support*, Rakendused mis pikemat aega uuendusi
Node	Klastris paiknev masin
NuGet	.NET pakethaldur

.NET	Microsofti poolt loodud raamistik
PHP	Veebirakednuste arendamisel kasutatav skriptimiskeel
PL	*Presentation Layer*, esitluskiht
Read-through	Vahemällu salvestamise strateegia, kus vahemälu küsib andmeallikast andmeid ja salvestab
Refresh-ahead	Vahemällu salvestamise strateegia, kus andmed on vahemällu eellaetud ja neid värskendatakse perioodiliselt
Rehostimine	Hõlmab olemasoleva tarkvara ja keskkonna viimist uuele infrastruktuurile, ilma et oleks vaja teha olemasolevale koodile suuri muudatusi
Repository (pattern)	Tarkvaraarenuses kasutatav muster, kus defineeritakse standardiseeritud meetodid andmeallikatele juurdepääsemiseks
REST	*Representational State Transfer*, arhitektuuristiil, kuidas süsteemid saavad HTTP vaheldusel suhelda
Reverse-proxy	Pöördproksi on vahendaja rollis asetsev teenus, mis võtab vastu kasutajate päringud ning suunab need tagarakendusele. Seejärel tagastab pöördpoksi teenus tagarakendusest saadud tulemused kasutajal.
Single Responsibility Principle	SOLID, tarkvaraarenduse printsiip, kus klassil tohib olla ainult üks põhjus muutmiseks
Skaleeritavus	Tegemist on süsteemiga mida saab laiendada, et see oleks võimeline toime tulema suurema töökoormusega
SPA	*Single Page Application*, üheleherakendus uuendab dünaamiliselt veebilehe sisu ilma, et oleks vaja tervet lehte uuesti alla laadida
SQL	*Structured Query Language*, andmebaasiga suhtlemise keel
Zero-downtime	Olukord, kus rakenduse juurutamise käigus ei katke teenus kliendile
TS	TypeScript, programeerimisekeel, mis võimaldab arendajatel kasutada tüübitud väärtuseid *JavaScript*'is
UI	*User Interface*, kasutajaliides
VM	Virtuaalmasin
Worker node	Klastris paiknev masin, mis tegeleb Dockeri konteinerite käivitamisega
XML	*Extensible Markup Language*, andmevahetusvorming
xUnit	Ühiktestide kirjutamisel kasutatav testide raamistik .NET tehnoloogiaatel

Sisukord

1 Sissejuhatus 11
2 Taust 12
2.1 Pärandsüsteem 12
2.2 Pärandsüsteemi migreerimise strateegiad 13
2.3 Rakenduse arhitektuur 14
2.3.1 Monoliitne arhitektuur 14
2.3.2 Mikroteenuste arhitektuur 14
2.4 Strateegiad vahemällu andmete laadimiseks 15
2.4.1 Cache-aside strateegia 15
2.4.2 Read-through 16
2.4.3 Refresh-ahead 16
2.5 Tehnoloogilised võimalused 17
2.5.1 .NET tehnoloogiatel põhinev tagarakendus 17
2.5.2 Esirakendus 17
3 Pärandsüsteemi analüüs 19
3.1.1 Pärandsüsteemi tutvustus 19
3.1.2 Pärandsüsteemi ebastabiilsuse tekitajad 19
3.1.3 Andmeallikaga suhtlemise struktuur pärandsüsteemis 20
3.2 Probleemi püstitus 20
4 Lahenduse analüüs 21
4.1 Nõuded lahendusele 21
4.2 Pärandsüsteemi migreerimine 22
4.3 .NET põhise tagarakenduse struktuur 22
4.3.1 Kasutatavad tehnoloogiad 22
4.3.2 Rakenduse arhitektuuriline disain 23
4.3.3 Kihilise arhitektuuri rakendamine tagarakenduses 23
4.3.4 Implementeeritav andmeallikatega suhtlus struktuur uues rakenduses 24
4.3.5 Pärandsüsteemi andmete migreerimine ja hoiustamine uues süsteemis 24
4.4 Esirakenduse tehnoloogiline valik 26
4.5 Vahemällu salvestavate strateegiate valik 26
5 Rakenduse arendusprotsess 28

5.1 Andmeallikaga suhtlemise projekti implementatsioon 28
5.2 Andmekihi implementeerimine 29
5.2.1 Cache-aside strateegia 30
5.2.2 Refresh-ahead strateegia 31
5.3 Äriloogikakihi implementeerimine 32
5.4 Esitluskihi implementeerimine ASP.NET Web API kontrolleritel 33
5.5 Esirakenduse arendus 34
5.5.1 Projekti struktuur 34
5.5.2 Esirakenduse interaktsioon segmenteeritud tagarakendusega 35
5.6 Uue süsteemi juurutamise protsess 36
5.7 Rakenduse testimine 37
6 Tulemuste analüüs 39
6.1 Uue süsteemi stabiilsus ja juurutamise protsess 39
6.2 Uue süsteemi jõudluse võrdlus pärand süsteemiga 40
6.3 Edasiarendused 40
7 Kokkuvõte 42
8 Kasutatud kirjandus 43
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 46

Jooniste loetelu

Joonis 1.*Cache-aside* strateegia 15
Joonis 2. *Read-through* strateegia 16
Joonis 3. *Refresh-ahead* strateegia 17
Joonis 4 Kihiline arhitektuur 23
Joonis 5. Näide andmeallikaga suhtlemise projekti struktuurist 29
Joonis 6. *HttpClient* registreerimine ASP.NET sõltuvuste süsteemi 29
Joonis 7. Näide MemoryCache kasutusest 31
Joonis 8. Perioodiline andmete värskendamine 31
Joonis 9. Näide *SermaphoreSlim* kasutusest 32
Joonis 10. Ühekihilised teenused 32
Joonis 11. Kahekihilised teenused 33
Joonis 12. Näide *ResponseCache* kasutusest 34
Joonis 13. Näide reverse-proxy kasutusest 36
Joonis 14. Axios päringu näide eraldiseisva tagarakendusega suhtlemisel 36
Joonis 15. Näide z*ero-downtime* ja *restart_policy* Docker Swarmi konfiguratsioonist 37

Tabelite loetelu

Tabel 1. Kahe süsteemi vaheline jõudluse võrdlus 40

1 Sissejuhatus

Tarkvaraarendusettevõttes on kasutusel teenuste süsteem, mis genereerib raporteid ja analüüse välistest andmeallikatest saadud andmete põhjal. Tegemist on pärandsüsteemiga millel on hakanud esinema jõudlus ja stabiilsus probleeme ning rakenduse edasised arendused on muutunud keerukaks ning häirib kasutajate igapäevast töövoogu. Käesoleva lõputöö eesmargiks on leida lahendus, mis hõlbustaks toote üleminekut olemasolevalt tehnoloogiliselt platvormilt .NET põhistelele tehnoloogiatele. Selle käigus luua uutel tehnoloogiatel põhinev, stabiilsem ja jõudlusprobleemideta toote alus.

Töö teises peatükis annab töö autor ülevaate pärandsüsteemi olemusest, selle probleemidest ja migreerimise strateegiatest. Seejärel tutvustatakse võimalikke tehnoloogilisi lahendusi, arhitektuurimudeleid ja andmesalvestus strateegiaid, kuidas võiks uus süsteem välja näha. Seejärel määratletakse uue süsteemi nõuded ja piiritletakse arendustöö skoop. Selle põhjal võrreldakse erinevaid tehnoloogiaid ja migreerimise strateegiaid ning valitakse parimad variandid uue süsteemi arendamiseks, mis lahendaksid pärandsüsteemis esinevad probleemid. Analüüsist tuginedes alustatakse arendusprotsessiga, mille käigus valmib uuel teenusel paiknev esi- ja tagarakendus ning teostatakse testimine ja juurutamine erinevates toote keskkondades. Lõpuks võrreldakse arenduse käigus valminud süsteemi pärandsüsteemiga, kontrollides valminud rakenduse stabiilsust ja jõudlust.

2 Taust

Selles peatükis tutvustab töö autor pärandsüsteemi olemust, süsteemi arhitektuuri, migreerimise strateegiaid ja andmete vahemällu laadimise strateegiaid. Samuti käsitletakse tehnoloogilisi valikuid ning tutvustatakse lõputöös käsitletavat pärandsüsteemi ja sellega kaasnevaid probleeme.

2.1 Pärandsüsteem

Pärandsüsteemiks saab nimetada riistvara, tarkvara või rakendusi, mis on endiselt kasutusel vaatamata uuematele ja tõhusamate alternatiivide olemasolule. Need süsteemid on tihtipeale ettevõtte tegevuse lahutamatuks osaks ning enam kui 66% organisatsioonidest toetub jätkuvalt pärandsüsteemidele [1]. Ent antud süsteemide jätkusuutlikus ja ajakohastamine võivad osutuda keeruliseks ning ressursimahukaks. [2]

Pärand süsteemi iseloomustavad järgmised aspektid [2]:

- Süsteemide kasutavad vananenud programmeerimise keeli, tarkvara või rakendusi
- Neid on raske integreerida kaasaegsete tehnoloogiatega
- Süsteemide haldamine ja uuendamine on kulukas
- Süsteemide ei toeta ettevõtte kasvu või uute funktsionaalsuste integreerimist

Vaatamata pärandsüsteemide uuendamisest tulenevatele potentsiaalsetele eelistele. Eelistavad mitmed ettevõtted jätkata olemasolevate süsteemide kasutamist. Kuna süsteemid on võimelised täitma neile antud ülesandeid ja uuele platvormile migreerumine võib kujuneda keeruliseks ning kulukaks protsessiks [2]. Mis nõuab detailset analüüsi ja testimist, et tagada uue süsteemi vastavuse ettevõtte äri nõuetele [3].

Üheks oluliseks ajendiks pärandsüsteemidelt uutele tehnoloogiatele migreerumisel on süsteemi ebapiisav jõudlus, mis seeläbi võib kahjustada nii töötajate produktiivsust kui

ka lõppkasutajate kogemust. Lisaks toob vananenud tarkvara kasutamine kaasa olulise turvariski rakendusele, millele enam ei väljastata turvauuendusi [4].

2.2 Pärandsüsteemi migreerimise strateegiad

Käesolev lõputöö keskendub igapäevaselt kasutatavate rakenduse tööriistade migreerimisele. seega on esmatähtis tagada teenuste pidev kättesaadavus ja sujuv üleminek uuele platvormile. Selleks on vaja koostada plaan mille alusel saaks olemasolevaid süsteeme edukalt ja optimaalselt üle tuua. Järgnevalt tutvustatakse lähemalt nelja pärandsüsteemi migreerimise strateegiat: rehostimise, refaktoriseerimise, asendamise ja ülesehitamise strateegiad.

Rehostimise strateegia hõlmab olemasoleva tarkvara ja keskkonna pilve migreerimist, ilma et oleks vaja teha olemasolevale koodile suuri muudatusi. Süsteem toimiks endiselt samamoodi nagu senini ning ei võta kasutusele pilvekeskkonnas pakutava paindlikkuse ja skaleeritavuse eeliseid. [4]

Refaktoriseerimine on protsess, mille käigus täiendatakse olemasolevat koodi, ilma et luuakse uut funktsionaalsust. Selle käigus saab muuta olemasoleva rakenduse puhta koodi põhimõtetele vastavaks, millel on lihtne disain, täiendatud jõudlus, skaleeritav ja paindlik. Projekti refaktoriseerimise ajal saab projekti viia üle tingimustele, et seda saaks hostida pilves. Mõnel juhul on mõistlik refaktoreerida rakendust järk-järgult. [4]

Asendamise strateegia hõlmab kogu olemasoleva pärandsüsteemi üleviimist uuele platvormile. See valik sobib siis kui platvorm on juba vananenud ja valmis lahendus on optimaalsem ning suudab täita vajalikku funktsionaalsust. [4]

Ülesehitamis strateegia nõuab kogu süsteemi ümberkirjutamist ja ehitamist. Seda strateegiat kasutatakse siis kui olemasolev rakendus ei suuda täita ärinõudeid ning vajab täielikku ülesehitamist, et toetada uusi funktsioone ja tehnoloogiaid. Arhitektuuri muutmine säilitab olemasoleva väärtusliku koodi ja asendab vananenud aluse, et luua rakendus, mis on vastupidavam, automatiseeritum, kättesaadavam ja turvalisem. [4]

2.3 Rakenduse arhitektuur

2.3.1 Monoliitne arhitektuur

Monoliitne arhitektuur on tarkvaraarenduses traditsiooniline mudel, kus rakendus on üles ehitatud tervikuna ja ei sõltu teistest rakendustest [5]. Rakenduse ei pea olema sisemiselt monoliitne, kuid see on struktureeritud erinevate komponentide, tasandite ja kihtidena, nagu näiteks rakenduse kiht, domeeni kiht ja andmejuurdepääsu kiht [6].

Kui tekib vajadus rakenduse võimekust või jõudlust suurendada, saab luua olemasolevast rakendusest uue koopia kas konteinerina või virtuaalmasinana (VM). Seejärel saab koormusjaoturi abil jagada liiklus mitme teenuse vahel, et vältida olukorda, kus üks teenus saab ülekoormatud. [6]

Sellise lähenemise puudused ilmnevad rakenduse kasvades ja skaleerimise vajaduse tekkimisel. Kui kogu rakendust on vaja skaleerida, nõuab see täiendavaid süsteemiressursse. Enamasti on vaja skaleerida vaid üksikuid ressursimahukaid teenuseid. [6]

2.3.2 Mikroteenuste arhitektuur

Erinevalt monoliitsest rakendusest on igal mikroteenusel hajutatud arhitektuuris üks kindel funktsioon või äriloogika, mida ta peab täitma. See tähendab, et mikroteenuse arhitektuur on peamiselt suunatud tagarakendusele, kuigi antud lähenemist on võimalik kasutada ka esirakenduses. Iga teenus töötab iseseisvalt ja suhtleb teiste konteineritega kas HTTP/HTTPS või AMQP protokollide kaudu. Igal mikroteenusel on kindel roll, mida täidetakse süsteemis ning seda peab saama arendada ja juurutada iseseisvalt. Lõpetuseks peaks igal mikroteenusel olema domeeni andmemudel ja domeeniloogika ning see võib põhineda erinevatel andmesalvestustehnoloogiatel ja erinevatel programmeerimiskeeltel. [6]

Võrreldes monoliitse rakenduse skaleeritavust mikroteenustega siis on võimalik igat mikroteenust iseseisvalt juurutada. Nii saame skaleerida ainult neid süsteemi komponente, mis vajavad täiendavat jõudlust, et tagada süsteemi kasutatavus. Kui monoliitset rakendust on vaja skaleerida siis skaleeritakse tervet rakendust. See tähendab, et lisaks vajalikele teenustele skaleeritakse teenuseid, mis ei vaja skaleerimist. [6]

2.4 Strateegiad vahemällu andmete laadimiseks

Vahemälu kasutatakse andmete salvestamiseks, millele juurdepääs võib osutuda ressursimahukaks või vajab sagedast ligipääsu. Kuna andmete lugemine vahemälust on märkimisväärselt kiirem kui nende hankimine algsest andmeallikast. Lisaks vähendab vahemälu kasutamine rakenduse ja andmebaasi vahelist suhtlust, minimeerides korduvate andmepäringute vajadust ja vabastades seeläbi andmebaasi ressursse teiste toimingute jaoks. Järgnevalt käsitletakse lähemalt vahemällu kirjutamist strateegiaid, mis on peamiselt mõeldud andmete kogumiseks.

2.4.1 Cache-aside strateegia

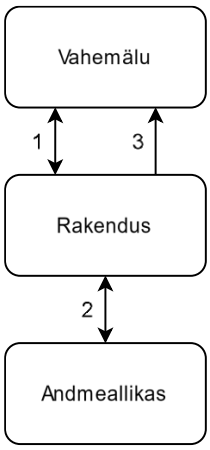
Tegemist on andmete vahemällu laadimise mustriga, mida tuntakse kui laisa laadimisena. Selle strateegia peamiseks eeliseks on see, et vahemällu salvestatakse ainult andmed mis küsitakse ja strateegiat on suhteliselt lihtne rakendada. Peamiseks nõrkuseks on potentsiaalne ebakõla vahemälu ja andmebaasis paiknevate andmete vahel. [7]

Andmete kogumine *cache-aside*(Joonis 1) strateegiaga toimub kolmes etapis:

1. Rakendus otsib andmeid kõigepealt vahemälust. Kui andmed leitakse vahemälust, tagastatakse need rakendusele. [8]

2. Kui vahemälu ei sisaldanud vajalikke andmeid siis rakendus küsib andmed andmeallikast ja tagastab need. [8]

3. Lõpetuseks laeb rakendus andmeallikast saadud andmed vahemällu, et tulevased päringud saaksid neile kiiremini juurde pääseda. [8]



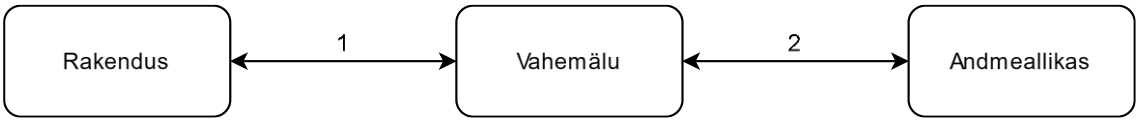
Joonis 1.*Cache-aside* strateegia

2.4.2 Read-through

Read-through strateegia (Joonis 2) korral toimub andmete lugemine läbi vahemälu ja vahemälu suhtleb andmebaasiga laisa laadimise põhimõttel. See strateegia delegeerib andmete vahemällu laadimise rakenduse asemel vahemälu teenusele. [9]

Andmete kogu mine read-through strateegiaga jaguneb kaheks etapiks:

1. Rakendus otsib andmeid vahemälust. Kui andmed leitakse siis tagastatakse need rakendusele. [8]
2. Kui andmeid ei leita vahemälust siis võtab vahemälu ühendust andmebaasiga ja salvesta need vahemällu ning seejärel tagastab andmed rakendusele. [8]



Joonis 2. *Read-through* strateegia

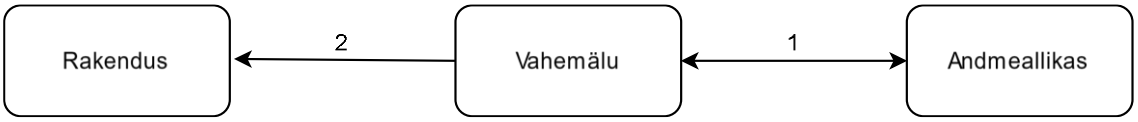
Peamine erinevus *read-through* ja *cache-aside* strateegiate vael seisneb rakenduse vastutuses. *Cache-aside* strateegiaga vastutab rakendus andmeallikaga suhtlemise ja vahemällu laadimise eest, aga *read-through* strateegia korral on delegeeritakse ülesanne vahemälule. Sarnaselt *cache-aside* strateegiale võib ka *read-through* strateegia korral esineda ebakõla vahemälus kui ka andmebaasis seinevata andmete vahel. [9]

2.4.3 Refresh-ahead

Refresh-ahead (Joonis 3) strateegia puhul on andmed vahemällu eellaetud ja värskendatakse enne nende aegumist. Strateegia peamiseks eeliseks on kiire ligipääs otsitavatele andmetele ja vähendatud võrgulatentsus. Samas on selle lähenemisviisi peamisteks puudusteks mälu ebaefektiivne kasutamine ja rakendamise keerukus. [10]

Andmete kogumine *Refresh-ahead* strateegiaga jaguneb kaheks etapiks:

1. Andmed on vahemällu eellaetud ja toimub perioodiline andmete uuendamine. [8]
2. Rakendus otsib andmed vahemälust. [8]



Joonis 3. *Refresh-ahead* strateegia

2.5 Tehnoloogilised võimalused

Järgnevas peatükis kirjeldatakse lähemalt tehnoloogilisi võimalusi, mida saab kasutada lõputöös uue süsteemi arendamiseks.

2.5.1 .NET tehnoloogiatel põhinev tagarakendus

Tagarakenduse arendamiseks kasutatakse Microsofti poolt loodud objektorienteeritud C# programmeerimise keelega [11]. See kuulub C-põhiste keelte perekonda ning jagab sarnasusi C, C++, Java ja JavaScriptiga [11]. Lisaks on C# tüübikindel keel, mida iseloomustab põhjalik dokumentatsioon ja ulatusliku pakettide valikuga [12].

.NET ökosüsteemis on kaks peamist raamistikku, mida kasutatakse C# rakenduste arendamiseks: .NET (varasemalt tuntud kui .NET Core) ja .NET Framework. Mõlemad raamistikud kasutavad rakenduse ehitamiseks .NET CLI ja sõltuvuste haldamiseks NuGet paketihalduri.

.NET (varasemalt tuntud .NET Core nimega) on Microsofti poolt arendatud avatud lähtekoodiga raamistik, mis jookseb nii Windowsi kui ka Linuxi-põhistel operatsioonisüsteemidel. Raamistik on optimeeritud pilvepõhiste rakenduste ja mikroteenuste käitamiseks ning on skaleeritav ja kõrge jõudlusega [13].

.NET Framework on Microsofti poolt loodud arendusplatvorm, mis on mõeldud Windowsi operatsioonisüsteemil paiknevate veebi- ja töölauarakenduste (Windows Forms, Windows Presentation Foundation) arendamiseks. Platvormi esmane versioon 1.0 lasti välja 2002. aastal ja on laialdaselt kasutuses ettevõttete pärandsüsteemides. [13] [14]

2.5.2 Esirakendus

Esirakendus toimib vahendajana kliendi ja tagarakenduse vahel, hõlmates esiliidese funktsionaalsuse arendust, mis on peamine interaktsiooni punkt kasutaja ja tagarakenduse

vahel. Järgnevalt tutvustatakse kahte populaarsemat esiliidese arendusraamistikku: Vue ja React.

Vue on Evan You poolt 2014. aastal loodud avatud lähtekoodiga progressiivne JavaScripti raamistik, mille eesmärk on ühendada Angular'i, Ember'i ja React'i parimad praktikad, muutes veebirakenduste arendamise kiiremaks ja mugavamaks [15]. Raamistik sobib hästi ühe lehe rakenduste (SPA) arendamiseks, kus see haldab lehe andmeliiklust ja navigeerimist [16]. Vue suurimateks eelisteks on selle lihtsus, paindlikkus, võimalus luua taaskasutatavaid komponente, kvaliteetne dokumentatsioon ja mugav integreerimine olemasolevate projektidega [15].

Raamistiku peamisteks nõrkusteks on väike inglisekeelne kogukond, mis võib piirata materjalide ja abi kättesaadavust. Lisaks võib liigne keeleline paindlikkus võib muuta koodi haldamise keerulisemaks ja põhjustada probleeme suuremates arendustöödes. [17]

React on Facebooki loodud komponendipõhine avatud lähtekoodiga JavaScripti teek, mis on tuntud oma tõhususe, korduvkasutatavate komponentide ja tugeva kogukonna poolest. Tänu oma populaarsusele ühildub React paljude kolmandate osapoolte teekidega. Lisaks veebirakenduste arendamisele võimaldab React Native luua rakendusi IOS ja Android platvormidele. Reacti peamisteks nõrkuseks on selle keskendumine vaid kasutajaliidese loomisele, mistõttu vajab see integreerimist kolmandate osapoolte teekidega, näiteks React Routeri ja Reduxiga ja sellega kaasneb ka õppimise keerukus. [18] [19]

3 Pärandsüsteemi analüüs

Antud peatükis tutvustab autor lähemalt lõputöös käsitletavat pärandsüsteemi rakendust ja sellega seonduvaid probleeme.

3.1.1 Pärandsüsteemi tutvustus

Automatiseeritud Teenuste Süsteem on tarkvaraarenduse ettevõtte siseveebis paiknev iseseisvate tööriistade kogum, mille peamiseks ülesandeks on koguda andmeid erinevatest andmeallikatest REST API liideste kaudu. Saadud tulemused analüüsitakse rakenduse tööriistade poolt, et genereerida kokkuvõtteid erinevate protsesside ning teenuste kohta. Kasutajad pääsevad ligi tulemustele läbi veebipõhise kasutajaliidese.

Pärandsüsteem on arendatud PHP programmeerimisekeeles ja kasutab MySQL relatsioonilist andmebaasi andmete salvestamiseks. Süsteem paikneb Windows 10 põhises virtuaalmasinas.

3.1.2 Pärandsüsteemi ebastabiilsuse tekitajad

Automatiseeritud Teenuste Süsteemi peamiseks probleemiks on keskkonna ebastabiilsus, mis häirib kasutajate igapäevast töövoogu ja teenuse kasutatavust. Ebastabiilsuse põhjustajaid saab jagada kahte kategooriasse: koodi disain ja hostmasin.

Pärandsüsteemi arendusprotsess on viinud olukorrani, kus uute funktsionaalsuste lisamine ja vigade parandamine on muutunud keeruliseks. Samuti muudab koodi komplektsus ja automaattestide puudumine muudatuste tegemise aeganõudvaks ning riskantseks, suurendades vigade tekkimise võimalust. Lisaks saab lihtsustada mitmete süsteemi tööriistade äriloogikat kasutades alternatiivseid andmeallikate päringuid. Samuti esineb olukordi, kus teatud tööriistade samaaegsel käivitamisel tekib globaalseid hangumisi ja tõenäoline *blocking I/O*. Kus klient esitab serverile ühenduse loomise taotluse siis blokeeritakse ühendust haldav lõim seniks kuni on lõpetatud andmete lugemine või kirjutamine [20]. Selle tulemusena võib rakendus jääda pikaks ajaks laetavasse olekusse või seni, kuni mõni protsess lõpetab töö ja vabastab blokeeritud ressursi.

Teiseks probleemiks on rakenduse hostmasin. Pärandsüsteem paikneb Windows 10 virtuaalmasinas. Selline paigutus raskendab süsteemide eraldamist ja skaleerimist. Lisaks

on hostikeskkond ressursimahukas ning sisaldab teenuseid (näiteks brauser), mis ei ole vajalikud pärandsüsteemi toimimiseks. Lisaks sellele nõuab Windows 10 uuendused masina taaskäivitamast, mis põhjustavad pärandsüsteemi lühiajalist *downtime*. Kuna uuenduste implementeerimiseks on vaja muuta operatsioonisüsteemi tuumas töötavaid DLL faile, süsteemi draivereid ja registris paiknevate väärtuste redigeerimist [21].

3.1.3 Andmeallikaga suhtlemise struktuur pärandsüsteemis

Pärandsüsteemis loodi iga andmeallikaga suhtlemiseks eraldiseisev klass, mis vastutas ühest allikast andmete kogumise eest. Selle tulemusena on tekkinud mahukad ja komplektsed klassid, mille loogika jaotatud paljude erineva meetodite vahel. Kuna rakendus on olnud pikka aega kasutusel, kasutavad mitmed andmekogumise klassid vananenud REST API päringuid, mis on iganenud ega ole enam ühilduvad andmeallika toote hilisemate versioonidega.

3.2 Probleemi püstitus

Lõputöös käsitletakse tarkvaraarendusettevõtte siseveebi rakendust, mille toimimisest sõltuvad mitmed ettevõtte äriprotsessid. Kuid rakenduse jõudlusprobleemid ja pärandkood takistavad igapäevast töövoogu kui ka uute funktsionaalsuste implementeerimist. Seega on tekkinud vajadus uuele süsteemile, mis tagaks süsteemi stabiilsuse, parema jõudluse ja paindlikkuse edasisteks arendusteks.

Lõputöö eesmärgiks on leida lahendusi, mis hõlbustaks toote üleminekut olemasolevalt tehnoloogiliselt platvormilt kaasaegsemale ja skaleeritavale platvormile. Lisaks otsitakse viise, kuidas jagada sarnast funktsionaalsust erinevate siseveebi tööriistade vahel, vältides korduvat arendustööd ja tagades süsteemi ühtsema toimimise. Lõpptulemusena valmistatakse uutel tehnoloogiatel töötav, stabiilsem ning jõudlusprobleemideta toote alus.

4 Lahenduse analüüs

Antud peatükis kirjeldatakse lõputöös käsitletavale rakendusele nõudeid ning uuritakse erinevaid tehnoloogilisi võimalusi pärandsüsteemi migreerimiseks. Analüüsi tulemusena tehakse valik sobivate tehnoloogiate osas ning määratakse kindlaks pärandsüsteemi migreerimise strateegia.

4.1 Nõuded lahendusele

Kuna tegemist on rakendusega, mis on kriitiline tarkvaraarendusettevõtte igapäevases toimimises siis on uurimustöö autorile antud ette nõuded millega on vaja arvestada. Sellest lähtuvalt peab lõputöös arvestama järgnvate nõuetega:

- Rakendus peab suutma genereerida raporteid ja analüüse kogutud andmete põhjal.
- Rakendus peab suutma koguda andmeid kolmandatest andmeallikates.
- Rakendus peab kasutama versioneeritud API päringuid.
- Lõppkasutajatel peab olema tagatud juurdepääs süsteemi poolt genereeritud tulemustele veebipõhise kasutajaliidese kaudu.
- Süsteem peab olema 30% kiirem kui pärandsüsteem.
- Automaatsed testid peavad katma 60%-70% rakenduse funktsionaalsusest.
- Rakendus peab paiknema Linuxi põhistel serveritel.
- Tagarakendus peab kasutama .NET platvormil põhinevaid tehnoloogiaid.

Arvestades rakenduse täielikuks ümberkirjutamiseks kulub aastaid on töö autor seadnud projektile piiratud skoopi. Lõputöö käigus on vaja üle tuua ärikriitilised teenused (näiteks raporti generaator). Lisaks luuakse uues süsteemis rakenduste arhitektuur ja keskkonna aluse, mis lihtsustab tulevikus teiste teenuste üleviimist uuele platvormile.

4.2 Pärandsüsteemi migreerimine

Võttes arvesse pärandsüsteemi disaini ja „Pärandsüsteemi migreerimise strateegiad“ peatükist mainitud migreerimise strateegiaid siis tasuks autoril kasutusele võtta segu ülesehitamise ja asendamise strateegiatest. Kuna tegemist on keeruka monoliitse rakendusega, kus erinevad tööriistad on iseseisvad, kasutatakse vananenud tehnoloogiad ja uute funktsionaalsuste lisamine ning vigade parandamine on muutunud keerukaks. Seega tasuks rakendus järkjärgult viia üle uutele tehnoloogiatele ja arhitektuurile. See võimaldab rakenduse sammhaaval üle toomist ja omakorda vähendaks koormust pärandsüsteemile, kuna järkjärguline funktsionaalsuste uute keskkonda liigutamine vähendab kasutajate arvu ja tehingute mahtu vanas süsteemis.

Samas antud valikuga kaasneb paar olukord, mis võimaldavad tekitada kasutajatele ebamugavust:

1. Keskkonnad paiknevad erinevatel veebisaitidel

2. Uue ja vana süsteemi disaini iseärasused

Selle probleemi lahendamiseks on võimalik lisada vanale veebisaidile uus funktsionaalsus. Kus vana sait ei sisalda enam otsest juurdepääsu olemasolevalt tööriistale. Vaid sisaldab *IFrame*’i, mille sees kuvatakse uuel süsteemil paiknevat tööriista, mille disain sarnaneb vanal süsteemil paikneva tööriistaga.

4.3 .NET põhise tagarakenduse struktuur

Selles peatükis analüüsitakse uue .NET põhise tagarakenduse struktuuri. Eesmärgiks on luua ülevaade kuidas on uus tagarakendus üles ehitatud ja kuidas lahendatakse pärandsüsteemi kitsaskohad.

4.3.1 Kasutatavad tehnoloogiad

Arvestades „2.5.1.NET tehnoloogiatel põhinev tagarakendus“ peatükis toodud asjaolu, et.NET Framework on suunatud Windowsi operatsioonisüsteemil paiknevate rakenduste arendamiseks. Otsustab töö autor kasutada tagarakenduse arendamiseks .NET Core raamistikku 8 LTS versiooni. Sellega tagatakse loodava süsteemi ühilduvus Linuxi-põhiste serveritega.

4.3.2 Rakenduse arhitektuuriline disain

Võttes arvesse „Rakenduse arhitektuur“ peatükis mainitud võimekusi ja erinevusi monoliitse ning mikroteenuste arhitektuuri vahel siis edaspidi kasutatakse lõputöö arenduses segu mõlemast arhitektuurist. Olemasolev süsteem on ise juba arendatud monoliitses arhitektuuris siis on näha, et kasvava süsteemi arendus võib aja jooksu muutuda komplektseks ja aeganõudvaks. Mikroteenuse eeliseid arvesse võttes on otstarbeks jagada pärandsüsteem mitmeks eraldiseisvaks teenuseks, et vähendada rakenduse komplektsust ning võimaldab süsteemi komponente iseseisvalt skaleerida. Pärandsüsteem koosneb tööriistadest millel puudub üksteise vaheline kommunikatsioon siis implementeeritakse uus süsteemi mikroteenuste põhimõttel. Kus süsteem on jaotatud väiksemateks teenusteks, aga teenuste vahel puudub suhtlus.

4.3.3 Kihilise arhitektuuri rakendamine tagarakenduses

Tagarakenduse arenduseks võetakse kasutusele n kihiline arhitektuur. Kus rakenduse loogika on eraldatud kolme kihi vahel (Joonis 4): Esitluskiht (PL), äriloogikakiht (BLL) ja andme kiht (DAL) [22].

Joonis 4 Kihiline arhitektuur

Kihiline arhitektuur tagab, et muudatused ühes kihis ei mõjutaks teisi kihte. Kui esitluskiht omaks otsest juurdepääsu andmekihile siis tekiks väga tihedalt seotud rakendus, kus komponentide vahel on palju sõltuvusi. Lisaks on iga kiht teineteisest kihiliselt sõltumatu ja omab piiratud teadmisi teiste kihtide toimimisest. [23]

Esitluskiht vastutab kliendi ja tagarakendsue vaheliste interaktsioonide haldamise eest [22]. Käesolev rakendus koosneb esitluskiht REST API kontrolleritest, mis võtavad vastu päringuid ja suhtlevad äriloogikaga kihiga (BLL) ning tagastavad andmed kasutajatele JSON formaadis.

Äriloogikakiht sisaldab rakenduse äriloogikat. Kus tegeldakse kasutajapäringute käsitlemise, andmetöötluse ja toimingutega, mis ei hõlma andmete salvestamist ega pärimist andmeallikast. [24]

Andmekihi peamiseks ülesandeks on olla vahendaja rollis BLL ja andmeallikate vahel [25]. Kust kiht vastutab andmete hankimise, salvestamise ja manipuleerimise eest [25]. Käesolevas rakenduses on implementeeritud *repository* ehk hoidlate mustrit, kus defineeritakse standardiseeritud meetodid andmeallikatele juurdepääsemiseks [26]. See lähenemisviis soodustab koodi taaskasutust ja parandab rakenduse hooldatavust.

4.3.4 Implementeeritav andmeallikatega suhtlus struktuur uues rakenduses

Uues süsteemis luuakse igale andmeallikale eraldiseisev projekt, mille peamiseks ülesanneteks on suhtlus kindla andmekogumiga ja andmeallikaga suhtlemise *HttpClient* päringud on loogiliselt jaotatud väiksemate implementatsioonide vahel. Kuna on tegemist eraldi seisva teegi projektiga siis saame importida andmeallikaga suhtlemise projekti teenustesse, mis vajavad kolmanda osapoole andmeid.

Kuna võib tekkida olukord kus ühest andmekogumist on saadaval mitu erinevat versiooni, millel võivad olla erinevate struktuuridega andmepäringud (nt REST API). Seega peab andmete kogumise projekt olema tagasi ühilduv eelnevate andmeallikate versioonidega. Selle alusel saab projekt sisemiselt jagada kolmeks kihiks: DTO, avalikud liidesed ja teenused. Kus kõik andmekihid on versioneeritud vastavalt andmeallika päringu disaini versiooniga. Seega kui samaaegselt kasutatakse mitut erinevat versiooni sama tüüpi andmeallikast siis saavad mõlemad tagaliidese rakendused korraga kasutada sama andmetega suhtlemise projekti.

4.3.5 Pärandsüsteemi andmete migreerimine ja hoiustamine uues süsteemis

Pärandsüsteem kasutab MySQL relatsioonilist andmebaasi, mis asub tagarakendusega samas virtuaalmasinas. Kuna tegemist on rakendusega mille peamiseks ülesandeks on koguda kokku andmeid erinevatest andmeallikatest siis pärandsüsteem kasutas andmebaasi rakenduse ja raportite konfiguratsioonide hoiustamiseks ning ainult andmebaasi administraatoril oli võimalik modifitseerida konfiguratsioone.

Arvestades andmebaasi kasutusotstarvet pärandsüsteemis siis on vaja uurida, kas tulevases süsteemis on otstarbekas kasutada eraldiseisvat andmebaasi konfiguratsioonide ja võimalusel ka kolmandate osapoolte andmete hoiustamiseks.

Peamiseks eeliseks andmebaasi kasutuseks oleks raporti konfiguratsioonide ja kolmandatelt osapooltelt saadud andmete hoiustamiseks. Tänu sellele saame muuta raportite genereerimist kiiremaks ja stabiilsemaks. Samuti saame vähendada võrgu liiklust ning kulusid, mis tulenevad kolmandatest osapooltelt andmete pärimisega. [27]

Muidugi kaasnevad andmete duplitseerimisega ka negatiivsed küljed. Esiteks kui on tegemist on andmetega, mis muutuvad pidevalt siis on vaja tagada veakindlus ja võimalus optimaalselt kontrollida, kas kolmandatelt osapooltelt saadud andmed on muutunud või mitte. Võimaluse korral saaksime kasutada kolmandate osapoolte REST API või andmebaasi välju, mis sisaldavad andmete uuendamise aegu.

Kui esineb erinevus implementeeritava teenuse ja kolmandate osapoole andmete vahel siis on vaja lokaalses andmebaasis värskendada kogutud andmeid. Samuti tekitab see märkimisväärset riski kui tekib ebakõla kahe süsteemi andmete vahel. Mis omakorda võib tekitada olukorra, kus rakendus võib genereerida vigaseid raporteid [27]. Selliste raportite põhjal võivad raporti kasutajad teha vigaseid äriotsuseid, mis põhinevad valedel andmetel [27]. Lisaks suurendab andmete dubleerimine regulatiivsete nõuete rikkumise ohtu, kuna andmete täpsus on kohustuslik igas valdkonnas. Samuti võivad ebaõiged andmed põhjustada nii finantsilist kui ka mainekahjustust.

Võttes arvesse eelnevaid tingimusi siis tasub uues süsteemis vältida andmete duplitseerimist ja kuna vanas süsteemis oli kasutatud andmebaasi ainult konfiguratsioonide hoiustamiseks siis uues on mõistlik hoiustada konfiguratsioone vahemälus või lasta konfiguratsioone hallata kolmanda osapoole teenustel. Sel juhul saame hoiustada raporti konfiguratsioone XML või JSON failidena Git'is. Kui on vaja lisada muudatusi konfiguratsioonidele siis jätab iga muudatus maha jälje koos põhjustega. Samuti vältimaks vigaste konfiguratsioonide sattumist teenusesse peavad konfiguratsioonid olema kaetud automaat testidega.

4.4 Esirakenduse tehnoloogiline valik

Esiliidese arenduseks ei ole töö autorile seatud piiranguid. Selle lähtuvalt kirjeldati 2.5.2 peatükis lähemalt kahte populaarsemat esiliidese arendusraamistikku. Kuigi mõlemad raamistikud on võimekad esiliidese arendamisel, otsustas töö autor kasutada Vue raamistikku. See otsus tugines järgmistel tingimustel:

- React on UI teek, mis vajab SPA arhitektuuriga rakenduste arendamisel lisateeke. Vue seevastu pakub suurema osa SPA põhivajadustest ametlike teekidena.
- Töö auto eelistab Vue põhist lähenemist komponentide arendamisel, kus iga komponent on struktureeritud HTML, CSS ja JS/TS sektsioonideks
- Vue kasutab vaadete kihi ehitamiseks HTML-malle, kus HTML, CSS ja JS kood on selgesti eraldatav. Kuigi Vue toetab JSX-i kasutamist, keskendub React peamiselt JSX-il põhinevatele vaatete ehitamiseks. Vue lähenemine esiliidese arendusele on paindlikum ja paljudele arendajatele tuttavam, mis omakorda lihtsustab koostööd nii arendajate kui ka disainerite vahel. [15]
- Töö autoril on olemas varasem kogemus Vue-ga esiliideste arendamisel.

4.5 Vahemällu salvestavate strateegiate valik

Pärandsüsteemi analüüsi tulemustest selgus, et süsteemi peamine eesmärk on koondada andmeid erinevatest teenustest ning genereerida nende põhjal raporteid. Arvestades, et pärandsüsteem ei ole andmete omanik ega oma võimalust nende manipuleerimiseks siis ei ole otstarbekas lisada rakendusele eraldiseisvat andmebaasi andmete duplitseerimiseks. Seega kasutatakse uues süsteemise *cache-ahead* ja *refresh-ahead* strateegiaid, mis võimaldavad efektiivselt hallata rakendust vahemälus või staatiliste konfiguratsioonidena rakenduse konteineris.

Pärandsüsteem saab suure osa andmeid lokaalsest võrgust paikenvatelt REST API teenustel. Seega kasutatakse *cache-aside* strateegiat vaid nende andmete salvestamiseks, mille kogumine välistelt teenustelt on ajamahukas, päritakse tihti või mille töötlemine on ressursimahukas.

Refresh-ahead strateegiat hakatakse kasutama staatiliste raportite konfiguratsioonide lugemiseks. Konfiguratsioone hoiustatakse Git’i projektis ja juurutamise hetkel lisatakse need Dockeri konteinerisse.

5 Rakenduse arendusprotsess

Tagarakendus arendatakse ASP.NET abil ja kasutakse C# programmeerimise keelt. Rakenduse arendamiseks kasutatakse JetBrainsi Rider IDEA-t.

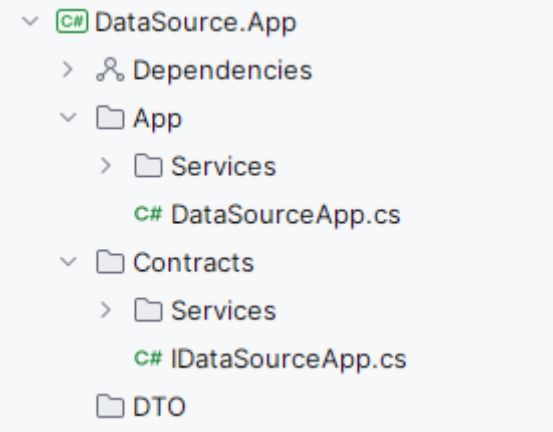
5.1 Andmeallikaga suhtlemise projekti implementatsioon

Kuna implmenteeritav rakendus ei ole andmete omanik ja vajab ligipääsu kolmandatele andmeallikastele siis implementeeritakse „Implementeeritav andmeallikatega suhtlus struktuur uues rakenduses" peatükis mainitud loogika ja luuakse igale andmeallikale eraldiseisev projekt. Mis sisaldab vajalikke REST API päringuid andmeallikaga suhtlemiseks. Antud lähenemisviis võimaldab koondada kokku ühe andmeallikaga seotud suhtlemise meetodid ühte taaskasutavasse projekti, mida oleks võimalik kasutada teistes teenustes. Projekti peamine kasutus hakkab paiknema iga rakenduse DAL-kihi *repository* klassis.

Projekti sisemine struktuur on jaotatud kolme kihi vahel (Joonis 5):

- DTO – Koosneb andmeedastusklassidest, mida kasutatakse andmete edastamiseks teistele rakenduse kihtidele
- Contracts – API teenuse avalikud liidesed
- Services – Sisaldab iga teenuse *Contracts* REST API implementatsiooni andmallikaga suhtlemiseks, andmete töötlemise ja DTO-de loomise eest.

Lisaks luuakse kõikide *Services* ja *Contracts* ligipääsemiseks eraldiseisev liidese, millesse on koondatud kõikide teenuste liidesed ja implementatsioonid. Sellist struktuuri saab nimetada fassaadimustriks, mis pakub lihtsustatud liidest keerukate alamsüsteemidele ligi pääsemiseks ja peidab süsteemi sisemisi detaile [33].



Joonis 5. Näide andmeallikaga suhtlemise projekti struktuurist

Seejärel registreeritakse andmeallikaga suhtlemise projekt ja *HttpClient* ASP.NET poolt pakutavas sõltuvuste süstimise süsteemi. Joonis 6 illustreerib *HttpClient* registreerimist ASP.NET sõltuvuste süsteemis

```
builder.Services.AddScoped<IDataSourceApp, DataSourceApp>();

builder.Services.AddHttpClient(HttpClientNames.DataSourceName.ToString
(), client =>
{
    client.BaseAddress = new Uri(userCredential.Url);
    client.DefaultRequestHeaders.Add(HeaderNames.Accept,
"application/json");
    client.DefaultRequestHeaders.Add("Authorization",
userCredential.Auth);
});
```

Joonis 6. *HttpClient* registreerimine ASP.NET sõltuvuste süsteemi

5.2 Andmekihi implementeerimine

DAL kihis rakendatakse „Andmeallikaga suhtlemise projekti implementatsioon“ peatükis kirjeldatud kihilist arhitektuuri, kus projekt on sisemiselt jaotatud kolme erineva struktuuri vahel: DTO, *Contracts* ja *Services*. Lisaks defineeritakse DAL kihis fikseeritud meetodid andmete otsimiseks ja andmeallikaga suhtlemiseks kasutatakse „Andmeallikaga suhtlemise projekti implementatsioon“ peatükis mainitud projekte.

Teatud konteineritega (nt raporti generaator) kaasatakse staatilised konfiguratsioonis siis osades *repository*-tes võetakse kasutusele *refresh-ahead* ja *cache-aside* strateegiad andmete vahemällu laadimiseks. Kus üks *repository* koosneb kahest osast: eelnevalt

defineeritud meetodid andemallikaga suhtlemiseks ja vahemälu funktsionaalsus, kuhu salvestatakse andmed *refresh-ahead* ja *cache-aside* strateegia alusel

5.2.1 Cache-aside strateegia

Cache-aside funktsionaalsuse implementeerimiseks saab võtta kasutusse .NET raamistiku paketi *Microsoft.Extensions.Caching.Memory*, mis sisaldab *IMemoryCache* liidest. Liides võimaldab hõlpsasti hallata andmete salvestamist ja manipuleerimist rakenduse vahemälus. Lisaks pakub see võimalust piirata vahemälule eraldatud mälumahtu ning andmete säilitamise kestust. *MemoryCache* initsialiseerimiseks ASP.NET REST API projekti *program.cs* failis kasutatakse käsku *builder.Services.AddMemoryCache()* ja Joonis 7 on toodud näide *MemoryCache* kasutusest koodis. [28]

```
private async Task<List<string>> GetComponents(Version source, Version
target)
 {
     var cacheKey = CacheKey(source, target);
     var cacheResult = _memoryCache.Get<IEnumerable<string>>(cacheKey);

    if (cacheResult != null)
     {
         return cacheResult.ToList();
     }

    var sourceComponentsAsync =
        await _appService
             .ReportCollectionService
             .GetComponentNamesAsync(source);
     var targetComponentsAsync =
        await _appService
             .ReportCollectionService
             .GetComponentNamesAsync(target);

    var result = sourceComponentsAsync.ToList();
     result
         .AddRange((targetComponentsAsync).ToList());
     result = result
         .DistinctBy(x => x).ToList();
     var cachedValue = _memoryCache
         .GetOrCreate(
         cacheKey,
         cacheEntry =>
         {
             cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(60);
             cacheEntry.AbsoluteExpirationRelativeToNow              =
TimeSpan.FromSeconds(120);
             return result;
         });
```

```
    return cachedValue!;
}
```

Joonis 7. Näide MemoryCache kasutusest

5.2.2 Refresh-ahead strateegia

IMemoryCache ei toeta *refresh-ahead* funktsionaalsuse implementeerimist ilma lisa arendusteta siis luuakse vahemällu salvestavate konfiguratsioonidele uus teenus. Kus iga konfiguratsioon saab endale eraldiseisva *repository* klassi, mille peamiseks ülesandeks on koondada kokku loogika, mis haldab andmete hoidmise loogikat vahemälus ja läbi mille saavad teised teenused ligipääsu vahemälus paiknevatele andmetele. Rakenduse juurutamise hetkel toimub andmete esmakordne mällu laadimine ja seejärel säilitatakse neid seal määratud aja (nt 15 minutit). Automaatse andmete värskendamiseks võetakse kasutusele *System.Time.Timer* (Joonis 8). Mille ülesandeks on teatud perioodi tagant värskendada andmeid [29].

```
_refresh = new System.Timers.Timer(refreshInterval);
_refresh.Elapsed += async (sender, e) => await LoadData();
_refresh.Start();
```

Joonis 8. Perioodiline andmete värskendamine

Arvestades, et teenusel on vaja ligipääsu raportite konfiguratsioonidele. Seega tasub hoida andmeid võti-väärtus paarides Rakenduse vahemälus. Selle jaoks kasutatakse *ConcurrencyDictionary*. Tegemist on andmetüübiga, mis on lõime kindel ja võimaldab mitmel lõimel samaaegselt hallata sõnastikkus paiknevaid andmeid, vältides seeläbi tõrkeolukordade tingimusi [30].

Andmete värskendamise ajal võib tekkida olukord, kus teised lõimed üritavad samal ajal andmeid lugeda [31]. Mis võib omakorda põhjustada vigaste või poolikute andmete tagastamist [31]. Selle probleemi lahendamiseks kasutatakse *Semaphoreslim* klassi (Joonis 9). Sellega saame tagada, et korraga tegeleks ainult üks lõim andmete haldamisega *ConcurrencyDictionary*'s, samal ajalt peavad teised lõimed ootama kuni eelnev protsess on enda töö lõpetanud [32].

```
private async Task LoadData()
{
    await _semaphore.WaitAsync();
    try
    {
        if (_internalJsonMapperDtoDataReaderOptimizer == null)
        {
```

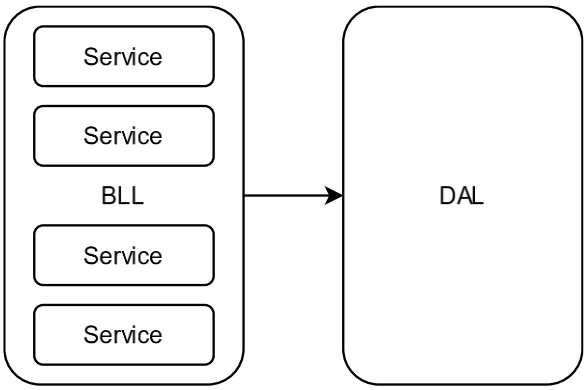
```
            _internalJsonMapperDtoDataReaderOptimizer =
                new
InternalJsonMapperDtoDataReaderOptimizer(_dataPath);
        }
        else
        {
            _internalJsonMapperDtoDataReaderOptimizer
                .UpdateData();
        }
    }
    finally
    {
        _semaphore.Release();
    }
}
```

Joonis 9. Näide *SermaphoreSlim* kasutusest

5.3 Äriloogikakihi implementeerimine

BLL kihi peamine eesmärk on DAL kihist saadud andmete töötlemine. Käesolevas rakenduses on rakendatud kahte tüüpi teenuseid: ühe- ja kahekihilised teenused.

Ühekihilised teenused (Joonis 10) on iseseisvad klassid, mis töötlevad DAL-ist saadud andmeid ilma täiendavate alamteenusteta. Ühekihilisi teenuseid kasutatakse juhul, kui andmete kogumine ja töötlemine ei nõua olulist ajakulu.

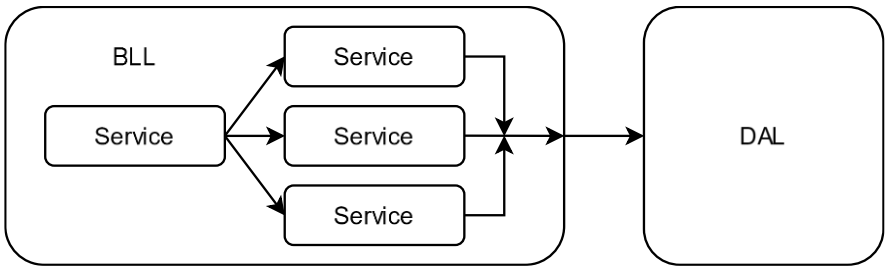


Joonis 10. Ühekihilised teenused

Kahekihilised teenused (Joonis 11) on teenused, mille toimimiseks on vaja alamteenuste abi. Teenused jaotatud kahte kategooriasse: pea- ja alusteenused.

- Peateenuse eesmärk on alusteenuste asünkroonne käivitamine ja nende väljundite koordineerimine.

- Alusteenused teostavad andmete kogumist ja töötlemist välistest andmeallikatest, mis võivad olla ajamahukad



Joonis 11. Kahekihilised teenused

Kahekihilise disaini peamiseks eeliseks on see, et paralleelselt täidetakse alamteenuste operatsioone ja tulemusi oodatakse ainult siis kui tekib nende järele vajadus, kasutades *await* ja *Task.WhenAll()* meetodeid [34]. See võimaldab peateenusel jätkata teiste äriprotsessidega ilma, et peaks ootama alamteenuse protsessi lõpetamist. Lisaks jälgivad alamteenused *Single Responsibility Principle*. Kus iga teenus täidab ühte spetsiifilist ülesannet ja lihtsustab üksikute komponentide testimist.

5.4 Esitluskihi implementeerimine ASP.NET Web API kontrolleritel

Kontrollerite arendamisel on järgitud API arhitektuuri. Tegemist on arhitektuuriga kus kasutatakse HTTP protokolli veebiteenustega suhtlemiseks. Kus iga päring on staatuseta ja töödeldakse iseseisvalt. See võimaldab klientidel kasutada standardseid HTTP meetodeid nagu GET, POST, PUT ja DELETE andmete pärimiseks ja manipuleerimiseks. Arhitektuuriline stiil tagab struktureeritud ja standardiseeritud meetodi teenuste vaheliseks suhtlemiseks. [35]

Rakendusse implementeeritakse versioonitud URL API, kus iga HTTP päring sisaldab versiooni *api/v1/reportGenerator/*. See tagab teenuste tarbijatele fikseeritud lõpp-punkti, mille alusel saab automatiseerida protsesse, mis on sõltumatud rakenduse hilisematest muudatustest [36]. Rakenduse versioneeritud päringute implementeerimiseks võetakse kasutusele *Asp.Versioning.Mvc.ApiExplorer* teek, mis võimaldab annotatsioonide abil versioneerida rakenduste päringuid.

Rakenduse kontrolleritele implementeeritakse vastuse vahemällu salvestamise funktsionaalsus *ResponseCache* abil, mille käigus hoitakse eelnevalt töödeldud päringu tulemus vahemälus. See võimaldab järgnevatele päringutele vastata otse vahemälust, ilma

et oleks vaja vastust uuesti genereerida. Selline lähenemine parandab oluliselt veebirakenduse jõudlust, eriti nende päringute puhul, mille genereerimine on kulukas ja mis ei muutu sageli. Joonis 12 illustreerib ResponseCache kasutamist kontrolleris. [36]

```
[HttpGet("/configurations")]
[ResponseCache(Duration = 360, Location = ResponseCacheLocation.Any,
NoStore = false)]
public async Task<ActionResult<string>> GetAllSources()
{
    List<string> allConfigurations = (await
_appService.ReportCollectionService.Get16MonthSources()).ToList();

    if (allConfigurations.Count == 0)
    {
        return NotFound( "Configutation sources were not found");
    }
    return Ok(allConfigurations);
}
```

Joonis 12. Näide *ResponseCache* kasutusest

5.5 Esirakenduse arendus

Esirakenduse arendamiseks kasutatakse Vue 3. Esirakenduse peamiseks ülesandeks on suhelda tagarakendusega ja kuvada tagarakendusest saadud tulemused kasutajatele loetavas formaadis. Esirakenduse arendamise jaoks kasutatakse JetBrainsi poolt loodud IntelliJ Ultimate IDEA-t.

Kasutajaliidese disainimiseks kasutatakse Vuetify raamistikku. Tegemist on kasutaja liidese disainimise raamistikuga, mis koosneb eelvalmistatud komponentidest. Vuetify jälgib Google Material disaini põhimõtteid tagades sellega ühtlase stiili ja on lihtne kasutusele võtta. [37] [38]

5.5.1 Projekti struktuur

Esirakenduse arendamisel on rakenduse loogika jaotatud alamkaustade vahel:

- *Components* – Kaust sisaldab rakenduse korduvkasutatavaid baas komponente
- *Helpers* – Kaust koondab kokku kõik globaalsed abimeetodid, mida kasutatakse rakenduse erinevates komponentides või vaadetes. Selle alla kuuluvad näiteks: globaalne otsing, aja teisendamine, failide alla laadimise loogika jne.

- *Services* – Kaust sisaldab loogikat mille alusel toimub tagarakendusega suhtlemine.

- *Stores* – Kaust sisaldab loogikat globaalese rakenduse oleku haldamise loogikat, mis on jagatud mitme vaate vahel. Näiteks paikneb antud kaustas globaalne teavituse haldamise süsteem.

- *Views* – Kaust on sisemiselt struktureeritud vaadete põhiselt. Kus iga vaate jaoks on loodud eraldiseisev alam kaust, mis hõlmab ainult vaate põhist loogikat ja taaskasutatavaid komponente

5.5.2 Esirakenduse interaktsioon segmenteeritud tagarakendusega

Uues rakenduses on tagarakendus segmenteeritud väiksemateks osadeks siis on vaja esiliidesel suhelda mitme erineva tagarakendusega. Ühe võimaliku lähenemiseviisina võib kaaluda iga tagarakenduse URLi defineerimist esirakenduses. Antud strateegia eeldab kõikide domeenide deklareerimist rakenduse juurutamise hetkel ja vajaks korduvat seadistamist, kui rakendus liiguks uute keskkonda.

Kuna eelnevalt mainitud protsess ei ole järjepidev ja võib olla ressursimahukas siis integreeritakse rakendus Nginx'iga. Nginx on kõrge jõudlusega avatud lähtekoodiga veebiserveri tarkvara, mis leiab laialdast kasutust *reverse-proxy*-na , koormusjaoturnina või vahemälu haldamisel [39]. Joonis 13 on näha näide reverse-proxy kasutusest, kus Nginx abil saame teha esiliidese ja tagaliidese kättesaadavaks samal domeenil, aga rakendused paiknevad eraldiseisvatel URL teedel. Sellise lähenemisega saame peita tagarakenduse sisemist struktuuri ja lihtsustada andmete pärimist esirakendusest.

```
location /
{
    try_files $uri $uri/ /index.html;
}

location /history/api/
{
    proxy_pass http://history:8080/api/;
    proxy_read_timeout 30s;
}

location /report-generation/api/
{
    proxy_pass http://reportgeneration:8080/api/;
```

```
    proxy_read_timeout 300s;
}
```

Joonis 13. Näide reverse-proxy kasutusest

Joonis 14 on toodud näide Axios päringust eraldiseisva tagarakendusega suhtlemisel.

```
export class ReportGenerationService extends BaseService {
constructor() { super("/report-generator"); }

async getReportConfiguration(): Promise<string[] | undefined> {
    try {
        const response = await
this.axios.get<string[]>('/configurations');

        if (response.status === 200) {
            return response.data;
        }

        return undefined;
    } catch (e) {
        console.error('error: ', (e as Error).message, e);
        throw e;
    }
}
```

Joonis 14. Axios päringu näide eraldiseisva tagarakendusega suhtlemisel

5.6 Uue süsteemi juurutamise protsess

Uues süsteemis on rakenduste juurutamine jaotatud kaheks etapiks.

Esimeses etapis luuakse Jenkinsi abil versioneeritud Docker image ja see laetakse sisevõrgu Docker repository-sse. Keskmiselt kulub uues süsteemis konteinerite ehitamiseks ja versioneeritud Docker image hoidlasse laadimiseks 131 sekundit.

Teises etapis toimub Dockeri konteineri juurutamine juba olemasolevatesse Linuxi põhistesse VM serveritesse Docker Swarmi abil. Docker Swarm on grupp füüsilisi või virtuaalmasinaid, mis töötavad ühes klastris [40]. Klastris paiknevaid masinaid nimetatakse *node*-teks [40]. Klaster koosneb vähemalt ühe juhtiva rolliga *node*-st, mis vastutab *worker node*-de tõhusa halduse ja toimimise eest [40].

Rakenduse konteineri juurutamisel otsustab juhtiv *node*, millisesse *worker node* paigutatakse konteiner ning vastutab selle edasise halduse eesti. Lisaks kasutatakse Docker Swarmi *update_config* funktsionaalsust. Kus „*order: start-first*“ parameetri

kasutusega saame tagada konteinerite järjepideva ülemineku vanalt konteineri versioonilt uuele [41]. Kus juurutamise hetkel töötavad korraga nii uue kui ka vana versiooniga konteiner [41]. Vana versiooniga konteiner suletakse Docker Swarmi poolt alles siis kui uus konteiner on saanud tervisliku staatuse. Sellise lähenemisega saame tagada rakendusele *zero-downtime* [41]. Lisaks võimaldab *restart_policy* parameetri konfigureerimine Docker Swarmil konteinereid automaatselt taaskäivitada rakenduse töö käigus ilmnevate tõrgete korral [42]. Joonis 15 on toodud näide zero-downtime ja restart_policy Docker Swarmi konfiguratsioonist.

```
deploy:
  update_config:
    parallelism: 1
    delay: 10s
    order: start-first

  restart_policy:
    condition: on-failure
    delay: 30s
    max_attempts: 10
```

Joonis 15. Näide z*ero-downtime* ja *restart_policy* Docker Swarmi konfiguratsioonist

5.7 Rakenduse testimine

Rakenduse testimise protsess on jaotatud kahte etappi: äriloogika testimine integratsiooni testide abil ning keskkonna spetsiifiline manuaalne testimine.

Äriloogika testimiseks on tagarakenduses implementeeritud xUnit-is arendatud integratsiooni testid. Testid kontrollivad rakenduse funktsionaalsuse korrektsust, et uute muudatuste integreerimise käigus ei kahjustata varasemalt loodud loogikat.

Keskkonna spetsiifiline testimine on jaotatud kahe keskkonna vahel:

- *Staging* – Simuleerib *production* keskkonda
- *Production* – lõppkasutajale kättesaadav keskkond

Enne muudatuste juurutamist *production* keskkonda on vaja läbida kvaliteedi kontroll *staging* keskkonnas. Iga muudatuse implementeerimiseks luuakse selle jaoks versioneeritud docker image, mis automaatselt juurutatakse *staging* keskkonda. Seejärel kontrollitakse *staging* keskkonnas implementeeritud muudatusi ja parandatakse testimise

käigus tuvastatud vead. Kui implementeeritav funktsionaalsus toimib *staging* keskkonnas ilma probleemideta siis migreeritakse muudatused *production* keskkonda, kus need muutuvad kättesaadavaks lõppkasutajale.

6 Tulemuste analüüs

Käesoleva lõputöö raames on välja töötatud .NET tehnoloogiatel põhinev tagarakenduse ja esirakenduse alus, mis asendab pärandsüsteemis paiknevad PHP põhiseid ärikriitilisi tööriistu. Loodud rakendus täidab kõik seatud nõuded ning uuel platvormil paiknevad teenused on edukalt võetud kasutusele ettevõtte igapäevases töös. Erinevalt pärandsüsteemist on uus süsteem kaetud 63% ulatuses automaattestidega. Sellega lihtsustatakse uute muudatuste tegemist, tagades et muudatuste käigus ei tekiks soovimatuid vigu.

Samuti tagavad versioonitud API-d teenuste päringute disaini stabiilsuse ja võimaldavad kasutajatel sujuvalt üle minna uutele API versioonidele. Ühtlasi on võetud kasutusele kihiline arhitektuur, mis loob aluse edasistele .NET tehnoloogiatel põhinevate tagarakenduse arendusele. Järgnevalt tutvustab töö autor lähemalt lõputöö käigus valminud süsteemi jõudlust ja stabiilsust.

6.1 Uue süsteemi stabiilsus ja juurutamise protsess

Pärandsüsteemi üks peamisi probleeme oli ebastabiilsus ja sagedased hangumised, mis häirisid igapäevast töövoogu Uue süsteemi testimise tulemused näitavad, et see on stabiilsem ja töökindlam. Segmenteeritud teenuste mustri kasutuselevõtuga saime eraldada toote loogika teineteisest, et vältida olukorda kus tooted võisid üksteist mõjutada ja lihtsustada toodete edasist arendustööd. Lisaks võeti kasutusele asünkroonne programmeerimise stiil, mis võimaldab mitmel protsessil paralleelselt töötada ning vältida olukorda, kus aeganõudvad protsessid võiksid üksteist blokeerida.

Kui pärandsüsteem töötas Windowsi põhises virtuaalmasinas siis uus süsteem on disainitud jooksma versioneeritud Dockeri konteineritel, mida saab juurutada Linuxi põhistesse serveritesse. Lisaks võimaldab Docker Swarmi kasutuselevõtt juhtival *node*-l hallata konteinerite haldamist, tagada konteinerite *zero-downtime* ning vea korral neid automaatselt taaskäivitada.

Lisaks võimaldas järkjärguline toodete üle viimine uuele süsteemile vähendada koormust pärandsüsteemil ja suunata see uuele platvormile. Mis omakorda aitas stabiliseerida vanas süsteemis paiknevate teenuste tööd.

6.2 Uue süsteemi jõudluse võrdlus pärand süsteemiga

Järgnevalt võrreldakse kahe süsteemi vahelist jõudlust, mille tulemusi kuvatakse Tabel 1. Jõudluse hindamiseks kasutatakse mõlemas süsteemis paiknevat raportite generaatorit. Testimiseks genereeritaks mõlemas rakenduses 1, 2 ja 6 kuu perioodiga raporteid. Mõlemad süsteemid paiknevad testimise ajal *production* keskkonnas.

Tabel 1. Kahe süsteemi vaheline jõudluse võrdlus

Periood kuudes	Aja kulu pärandsüsteemis (s)	Aja kulu uues süsteemis (s)
1	177.6	46.58
2	197.4	55.74
6	292.4	132.5

Testimis tulemused näitavad, et uues süsteemis on raportite genereerimise kiirus märkimisväärselt suurem võrreldes pärandsüsteemiga. Jõudluse paranemise peamiseks põhjuseks võib pidada üleminekut segmenteeritud arhitektuurile ja BLL kihis mainitud asünkroonsete alamteenuste kasutusele võtuga.

6.3 Edasiarendused

Käesoleva lõputöö fookuses on pärandsüsteemi migreerimine PHP põhiselt platvormilt .NET põhistele tehnoloogiatele. Kuna migreerimise protsess pole täielikult lõpule viidud, saab töö autor jätkata toodete järkjärgulist migreerimist uutele tehnoloogiatele ja kasutada ära uue süsteemi segmenteeritud arhitektuuri, tänu millele saab kasutada JVM-põhiseid tehnoloogiaid järgmiste toodete migreerimisel. Otsus põhineb asjaolust, et juba varasemalt on tehtud valmis JVM-i põhised moodulid, mis lihtsustavad kolmandate andme allikatega suhtlemist. Pärast kõikide toodete uuele süsteemile üleviimist saab alustada pärandsüsteemi täieliku eemaldamisega.

Kuna lõputöös võeti kasutusele järkjärguline teenuste üleviimine pärandsüsteemilt uuele ja mõlemad süsteemid töötasid paralleelselt. Seetõttu sarnaneb uue süsteemi disain pärandsüsteemiga, et kasutajatel oleks mugavam mõlemat keskkonda samaaegselt kasutada. Seega tasub tulevikus võtta ette arendustöid kasutajaliideste kasutatavuse

parandamiseks. Samuti kasutajatega vesteldes selgus, et nad sooviksid näha uuel süsteemil tumeda tooniga kasutajaliidest.

7 Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli uuendada tarkvaraarenduse ettevõtte pärandsüsteemi PHP põhiselt platvormilt .NET põhistele tehnoloogiatele. Selle ajendiks oli pärandsüsteemil esinevad jõudlus ja ebastabiilsuse probleemid, mis takistasid kasutajate igapäevast töövoogu ja uute funktsionaalsuste implementeerimist.

Töö käigus analüüsiti pärandsüsteemi kitsaskohti, migreerimise strateegiaid ja arhitektuurilisi valikuid. Parimaks lähenemisviisiks valiti teenuste järkjärguline üleviimine uuele platvormile, kus jagati pärandsüsteemis paiknevad teenused funktsionaalselt väiksemateks ja iseseisvateks tagarakendusteks.

Käesoleva töö käigus loodi .NET tehnoloogiatel põhinev tagarakenduse ja Vuel põhinev esirakenduse alus, mille tulemusel on uus süsteem stabiilsem ja jõudlusprobleemideta. Samuti on uus süsteem integreeritud ettevõtte igapäevasesse töövoogu, kus see asendab pärandsüsteemis paiknevaid ärikriitilisi tööriistu.

Lõputöö autor plaanib jätkata toodete järk-järgulist migreerimist uutele tehnoloogiatele, kasutades ära uue süsteemi segmenteeritud arhitektuuri, et edasiste toodete arendamisel kasuta JVM-i põhiseid tehnoloogiaid. Põhjuseks on see, et varasemalt on välja töötatud JVM-i moodulid, mis lihtsustavad andmevahetust kolmandate osapoolte andmeallikatega.

8 Kasutatud kirjandus

[1] A. Jaffery, „Legacy System: Definition, Challenges, Types & Modernization,“ 11.07.2024. [Võrgumaterjal]. Available: https://www.astera.com/type/blog/legacy-system/.. [Kasutatud 12.02.2025].

[2] C. Hetherington, „Understanding legacy system migration: how does it work?,“ 06.06.2024. [Võrgumaterjal]. Available: https://www.alumio.com/blog/legacy-system-migration-how-does-it-work. [Kasutatud 05.03.2025].

[3] A. Davis , „Learning Legacy Systems Migration Inside and Out,“ 14.01.2025. [Võrgumaterjal]. Available: https://www.openlegacy.com/blog/legacy-systems-migration. [Kasutatud 12.02.2025].

[4] I. Luchaninov, „6 Steps of Legacy System Migration to Balance Innovation and Continuity,“ 14.03.2025. [Võrgumaterjal]. Available: https://mobidev.biz/blog/legacy-system-migration-guide-best-practices-process-case-studies. [Kasutatud 12.02.2025].

[5] C. Harris , „Microservices vs. monolithic architecture,“ [Võrgumaterjal]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. [Kasutatud 07.02.2025].

[6] C. d. l. Torre, M. Rousos ja B. Wagner, .NET Microservices Architecture for Containerized .NET Applications, Microsoft Developer Division, .NET and Visual Studio product teams, 2023 .

[7] N. Kim, „Top 5 Caching Patterns,“ 05.10.2023. [Võrgumaterjal]. Available: https://newsletter.systemdesign.one/p/caching-patterns. [Kasutatud 18.02.2025].

[8] Shashir, „Overview of Caching, Distributed Cache, Caching Patterns & Techniques,“ 26.03.2022. [Võrgumaterjal]. Available: https://medium.com/geekculture/overview-of-caching-distributed-cache-caching-patterns-techniques-6130a116820. [Kasutatud 18.02.2025].

[9] S. Gautam, „Caching Strategy: Read-Through Pattern,“ [Võrgumaterjal]. Available: https://www.enjoyalgorithms.com/blog/read-through-caching-strategy. [Kasutatud 15.04.2025].

[10] S. Gautam, „Refresh Ahead Caching Strategy,“ [Võrgumaterjal]. Available: https://www.enjoyalgorithms.com/blog/refresh-ahead-caching-pattern. [Kasutatud 15.04.2025].

[11] „Introduction to C#,“ GeeksforGeeks, [Võrgumaterjal]. Available: https://www.geeksforgeeks.org/introduction-to-c-sharp/. [Kasutatud 09.02.2025].

[12] „Introduction to C#: definition and utilities,“ mytaskpanel, [Võrgumaterjal]. Available: https://www.mytaskpanel.com/introduction-to-csharp/. [Kasutatud 16.04.2025].

[13] „NET vs. .NET Framework for server apps,“ 11.11.2024. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server. [Kasutatud 05.04.2025].

[14] C. Sheladiya, „.NET Framework Vs .NET Core – A Fundamental Comparison!,“ 22.04.2022. [Võrgumaterjal]. Available: https://www.metizsoft.com/blog/net-core-vs-net-framework. [Kasutatud 05.04.2025].

[15] B. Cepil, P. Curac-Dahl, M. Klimowicz, N. Leśniak ja E. Neumann-Pluta, „State of Vue 2021 report,“ Monterail, [Võrgumaterjal]. Available: State of Vue 2021 report. [Kasutatud 09.04.25].

[16] „Introduction,“ [Võrgumaterjal]. Available: https://vuejs.org/guide/introduction.html. [Kasutatud 09.04.25].

[17] „What is Vue.js? Vue.js Advantages And Disadvantages,“ 22.11.2023. [Võrgumaterjal]. Available: https://foreignerds.com/vue-js-pros-and-cons/. [Kasutatud 09.24.25].

[18] „Advantages and Disadvantages of React JS,“ React Masters, 30.10.2023. [Võrgumaterjal]. Available: https://medium.com/@reactmasters.in/advantages-and-disadvantages-of-react-js-e6c80b25763b. [Kasutatud 09.04.2025].

[19] A. T. Williams, „The Pros and Cons of Using React Today,“ 04.12.2023. [Võrgumaterjal]. Available: https://thenewstack.io/the-pros-and-cons-of-using-react-today/. [Kasutatud 11.04.25].

[20] R. Athapathu, „Blocking I/O and non-blocking I/O,“ 31.08.2017. [Võrgumaterjal]. Available: https://medium.com/coderscorner/tale-of-client-server-and-socket-a6ef54a74763. [Kasutatud 13.04.2025].

[21] H. Liang ja . S. Xu , „Why you may be prompted to restart your computer after you install a security update on a Windows-based computer,“ [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/troubleshoot/windows-server/installing-updates-features-roles/why-prompted-restart-computer. [Kasutatud 15.02.2025].

[22] D. F. S. Dev, „Layered (N-Tier) Architecture in .NET Core,“ 08.10.2024. [Võrgumaterjal]. Available: https://dev.to/dotnetfullstackdev/layered-n-tier-architecture-in-net-core-51ic. [Kasutatud 14.04.2025].

[23] „Software Architecture Patterns,“ oreilly, [Võrgumaterjal]. Available: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html. [Kasutatud 14.04.2025].

[24] M. Ozkaya, „Layered (N-Layer) Architecture with SOLID Design Principles,“ 15.2.2023. [Võrgumaterjal]. Available: https://medium.com/design-microservices-architecture-with-patterns/layered-n-layer-architecture-with-solid-design-principles-15967a518ff1. [Kasutatud 14.04.2025].

[25] F. Salatino, „A Complete Guide to Master Data Access Layer Efficiently,“ 14.11.2024. [Võrgumaterjal]. Available: https://www.cdata.com/blog/data-access-layer. [Kasutatud 14.05.2024].

[26] P.-E. Bergman, „Repository Design Pattern,“ 20.04.2017. [Võrgumaterjal]. Available: https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30. [Kasutatud 15.04.2025].

[27] M. Chen , „Data Duplication Implications and Solutions,“ 04.09.2024. [Võrgumaterjal]. Available: https://www.oracle.com/ee/data-duplication. [Kasutatud 27.03.2025].

[28] „Design Patterns - Facade Pattern,“ tutorialspoint, [Võrgumaterjal]. Available: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm. [Kasutatud 14.04.2025].

[29] . R. Anderson, J. Luo ja . S. Smith, „Cache in-memory in ASP.NET Core,“ 24.04.2024. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-8.0. [Kasutatud 17.02.2025].

[30] „How to make System.Timers.Timer call async function synchronously when elapsed?,“ 28.03.2024. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/answers/questions/1635366/how-to-make-system-timers-timer-call-async-functio. [Kasutatud 26.02.2025].

[31] Rasul, „Understanding of ConcurrentDictionary in .NET,“ 08.05.2024. [Võrgumaterjal]. Available: https://resulhsn.medium.com/understanding-of-concurrentdictionary-in-net-3434105ba371. [Kasutatud 26.02.2025].

[32] H. Sharma, „Race Conditions: The Hidden Danger Of Multithreaded Programming,“ 9 12 2022. [Võrgumaterjal]. Available: https://harshit-28.medium.com/race-conditions-the-hidden-danger-of-multithreaded-programming-1e95577fb30c. [Kasutatud 02.03.2025].

[33] D. F. S. Dev, „.NET Pro Tip: Use SemaphoreSlim for Thread Synchronization,“ [Võrgumaterjal]. Available: https://dotnetfullstackdev.medium.com/net-pro-tip-use-semaphoreslim-for-thread-synchronization-afc17b4a308f.

[34] B. Wagner, B. Harvey , G. Warren ja E. Gritsina, „Asynchronous programming with async and await,“ 13.03.2025. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/.

[35] „Comprehensive Guide for Understanding REST APIs,“ Kong, 02.11.2024. [Võrgumaterjal]. Available: https://konghq.com/blog/learning-center/what-is-restful-api. [Kasutatud 18.04.2025].

[36] J. Shethiya, „Response Cache Attribute in .NET Core 8: Usage, and Examples,“ 23.09.2024. [Võrgumaterjal]. Available: https://medium.com/@jaimin_99136/response-cache-attribute-in-net-core-8-usage-and-examples-f32071c8f928. [Kasutatud 15.04.2025].

[37] „„What is Vuetify?,“ [Võrgumaterjal]. Available: https://vuetifyjs.com/en/introduction/why-vuetify. [Kasutatud 16.04.2025].

[38] „Why Vuetify,“ [Võrgumaterjal]. Available: https://vuetifyjs.com/en/introduction/why-vuetify. [Kasutatud 16.04.2025].

[39] S. Guides, „Nginx,“ [Võrgumaterjal]. Available: https://www.papertrail.com/solution/guides/nginx/. [Kasutatud 16.05.2025].

[40] „Docker Swarm - definition & overview,“ Sumo Logic, [Võrgumaterjal]. Available: https://www.sumologic.com/glossary/docker-swarm/. [Kasutatud 08.04.2025].

[41] D. Jovanović, „Self-host - Part 2 - Zero-Downtime Deployment using Docker Swarm,“ 18.07.2024. [Võrgumaterjal]. Available: https://dev.to/davorj94/self-host-part-2-zero-downtime-deployment-using-docker-swarm-2o3c. [Kasutatud 08.04.2025].

[42] „Compose Deploy Specification,“ docker, [Võrgumaterjal]. Available: https://docs.docker.com/reference/compose-file/deploy/. [Kasutatud 08.04.2025].

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

Mina, Karel Vuks

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Tarkvaraarendusettevõtte PHP-põhise pärandsüsteemi üleviimine .NET tehnoloogiatele", mille juhendaja on German Mumma
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

20.04.2025

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