TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Tarvi Raun 202819IADB

Eesti määrustele vastav kvaliteedijuhtimissüsteemi dokumentide heakskiitmine

Bakalaureusetöö

Juhendaja: German Mumma
Magistrikraad
Kaasjuhendaja Marion Lepmets
Doktorikraad

Tallinn 2023

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: Tarvi Raun

15.05.2023

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on luua pilootprojektina mikroteenuste arhitektuuril rajanev API-põhine rakendus, mis Eesti kontekstis pakub teenuseid meditsiiniseadmete tootmisvaldkonna kvaliteedijuhtimissüsteemitele dokumentide tõendatud digitaalseks allkirjastamiseks. Loodav API-põhine rakendus on eeskätt mõeldud integreerimiseks tarkvaraplatvormiga Confluence Cloud, millele lisatava funktsionaalsuse kaudu lahendatakse eksisteerivate lahenduste puudujääke. Rakendus ei ole avatud lähtekoodiga ning tehakse kättesaadavaks ainult vajalikele isikutele.

Arendusprotsessi käigus luuakse pilootprojektina API-põhine rakendus, mis võimaldab kasutajatel leida, muuta ning luua uusi allkirjastatavaid dokumente, moodustada ning leida allkirjastamiseks konteinereid ning allkirjastada dokumente. Arendus on jagatud mitmesse ossa, mis käsitlevad mikroteenuste arhitektuuri ja liidestamist, tagarakenduse ligipääsu lüüside ja teenuste ehitamist ning serverisse paigaldust.

Töö rõhk on API-põhise rakenduse arendamisel, mis võimaldab testida tarkvara teostatavust, sobivust ning annab aluse jätkutegevuste planeerimiseks. Tuleviku perspektiivis võimaldab rakendus koguda tagasisidet, mis annab aluse minimaalse töötava toode arendamiseks. Muuhulgas katab lõputöö kasutatuid tehnoloogiaid, arhitektuuri, testimist ning sellega seonduvad pidevat integratsiooni. Arendusprotsessi tulemuseks on pilootprojektina testitav API-põhine rakendus, mis põhineb mikroteenuste arhitektuuril.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 75 leheküljel, 6 peatükki, 9 joonist, 16 tabelit, 13 koodinäidet.

Abstract

Compliant Quality Management Document Approvals in Estonia

The purpose of this bachelor's thesis is to create an API-based application based on microservices architecture as a pilot project, which in the Estonian context offers services for quality management systems in the field of medical equipment production for certified digital signing of documents. The created API-based application is primarily intended for integration with the Confluence Cloud software platform, to which the shortcomings of existing solutions are solved through added functionality. The application is not open sourcecode and is made available only to the necessary persons.

During the development process, an API-based application is created as a pilot project, which allows users to add, find and create new documents to be signed, create and find signing containers and sign documents. The development is divided into several parts, which deal with the architecture and articulation of microservices, the construction of client application access gateways and services, and installation on the server.

The emphasis of the work is on the development of the service, which allows testing the suitability of the software and provides a basis for planning follow-up activities. In the perspective of the future, the application allows collecting feedback, which provides a basis for the development of a minimum working product. Among other things, the thesis covers the technologies used, architecture, testing and related continuous integration. The result of the development process is an API client application based on a microservices architecture that is being tested as a pilot project.

The thesis is in Estonian and contains 75 pages of text, 6 chapters, 9 figures, 16 tables, 13 code examples.

Lühendite ja mõistete sõnastik

aktor	süsteemi kasutajat esindav roll, mis kirjeldab süsteemis aktori poolt läbiviidavaid tegevusi
API	*Application Programming Interface* – rakendusliides
AJAX	*Asynchronous Javascript and XML* – tehnoloogiate kogum, mis võimaldab asünkroonseid päringuid
backend	teenusepoolne keskkond
Base64	kodeerimismeetod, mis teisendab binaarandmeid ASCII tekstiks ja vastupidi.
Bash	*Bourne Again Shell* – käsureaprotsessor UNIX'ile.
Body	API päringu keha ehk sisu
CI	*Continuous Intergration* – pidev integratsioon
CLI	*Command-line interface* – käsurea liides
CNAME	*Canonical Name Record* – DNS'i ressurss, mis täpsustab ühte domeeni teise aliasena
CORS	*Cross-origin resource sharing* – mehhanism, mis lubab domeenivälist infovahetust
CSS	*Cascading Style Sheets* – veebilehtede küljendamisel kasutatav märgistuskeel
CRUD	*Create, Read, Update, Delete* – püsiva salvestamise neli põhitoimingut
Dependency Injection	Sõltuvuste sisestamine – arhitektuuriline disanimuster, mis näeb ette objektide loomise eraldamist programmi käitumisest
DevOps	*Development and Operations* – praktikat, kus halduse ja arenduse insenerid töötavad koos terve teenuse elutsükli jooksul
digitaalne allkiri	Digitaalne signeerimisviis, mis on seaduslikult kehtiv ning juriidiliselt võrdne omakäelise allkirjaga
digiallkiri	Sama, mis digitaalne allkiri
DBMS	*Database Management System* – andmebaasi haldussüsteem
DNS	*Domain Name System* – detsentraliseeritud nimetussüsteem seadetele, mis on ühendatud võrguga
DTO	*Data Transfer Object* – andmeedastusobjekt

eIDAS	*electronic IDentification, Authentication and trust Services* – Euroopa Liidus kehtiv e-identimise ja e-tehingute määrus
Elektroonine allkiri	Kõik elektroonilisel viisil loodud allkirjad
ERD	*Entity Relationship Diagram* – olemi-suhte diagramm, metoodika andmemudelite koostamiseks ja kirjelduse esitamiseks
E2E	*End–to–End* – Lõppkasutustest
FURPS	Tarkvaranõuete klassifitseerimise mudel
frontend	Kasutajaliides, kliendipoolne keskkond
GDPR	*General Data Protection Regulation* – Euroopa Liitu isikuandmete kaitse üldmäärus
gRPC	*Google Remote Procedure Call* – Google poolt loodud laiendus RPC protokollile
HMAC SHA256	*Levinud räsifunktsioon*
HTML	*Hyper Text Markup Language* – veebilehe märgendamise keel
HTTP	*Hyper Text Transfer Protocol* – protokoll teabe edastamiseks arvutivõrkudes
IAC	*Infrastructure as Code* – kõrgetasemeline kodeerimiskeel, mis võimaldab IT-infrastruktuuri kirjeldada deklaratiivse koodina
IaaS	*Infrastructure as a Service* – pilvepõhine teenus, mis haldab infrastruktuurilisi elemente nagu server
IDE	*Integrated Development Environment* – rakendus, mis pakub arendajatele keskkonda koodi kirjutamiseks
IDL	*Interface Definition Language* – liideste määratluskeel, mille kaudu määratletakse andmestruktuuriskeemid ja programmeerimisliidesed
instants	Eksemplar – sarja element kontekstist sõltuvas tähenduses
ISO	*International Organisation for Standardisation* – rahvusvaheliste standarditega tegelev organisatsioon
JSON	*Javascript Object Notation* – Javascript'il põhinev andmevahetusvorming, mis võimaldab inimloetavalt kirjeldada objekte ja teisi vastavaid andmestruktuure.
JSX	*Eelprotsessor, mis lubab kasutada XML süntaksit Javascript'i koodis*
JWT	*JSON Web Token* – JSON-põhine standard ligipääsu sõnede koostamiseks

konteiner	Rakenduse standardne pakend, mis sisaldab programmikoodi, sõltuvusi, teeke või muid abivahendeid, mis rakenduse käivitamiseks vajalik
LTS	*Long-Term Support* – tehnoloogia pikaajaline tuge
metadata	Metaandmed, andmed, mis kirjeldavad teisi andmeid
MoSCoW	Prioritiseerimise prioritiseerimistehnika – lühend sõnadest *Must have*, *Should have*, *Could have*, and *Would like* but won't get
MDR	*Medical Device Regulation* – Euroopa Liidus meditsiiniseadme tootjate regulatsioon
NPM	*Node Package Manager* – Javascript'i teekide haldaja
online	Võrguga seotus
On-Premises	IT-infrastruktuur või riistvara, mida omab ja haldab ettevõtte Lokaalselt
ORM	*Object-Relational Mapping* – põhimõte, mille eesmärk on objekt-orienteeritud koodi kaardistada ümber teise andmevormi
QMS	*Quality Management Software* – kvaliteedijuhtimissüsteemi
PaaS	*Platform as a Service* – pilvepõhine teenus, mis haldab infrastruktuurilisi elemente, mis ei ole seotud rakendusega
Platvorm	Infotehnoloogias arvuti riistvara arhitektuur ja tarkvaraline baas, mis koos võimaldavad kasutada rakendustarkvara
POC	*Proof of concept* ehk pilootprojekt - meetodi või lahenduse realiseerimine tõestamaks teostatavust
production environment	Keskkond, mida kasutavad reaalsed kasutajad
plugin	lisandprogramm, mis on hõlpsalt installitav ja laiendab senise programmi või rakenduse võimalusi
REST	*Representational State Transfer* – veebiteenusega suhtlemise liidese arhitektuur
Restful	Veebirakendus, mis API järgib REST põhimõtteid
runtime	Käitusaeg, programmi jooksutamise aeg
SaaS	*Software as a Service* – pilvepõhine teenus, mille puhul on nii platvorm kui ka funktsionaalsused hallatud teenuse poolt
SES	*Simple Electronic Signature* – lihtne elektrooniline allkiri
Skript	Automatiseeritult teostatavaid toiminguid kirjeldav käsujada
SOA	*Service Oriented Architecture* – teenustele orienteeritud tarkvara arhitektuur
SOAP	*Simble Object Access Protocol* – veebiteenusega suhtlemise liidese protokoll

staging environment	Testimiskeskkond
tarkvaraplatvorm	tarkvarapõhine süsteemide ja teenuste tehnoloogiline alus
technology stack	tehnoloogiline kihtarhitektuur rakenduse või lahenduse moodustamiseks
Transponder	Riistvaraline või tarkvaraline kiht, mis vastutavad sõnumite edastamise eest erinevate süsteemide vahel.
Transpiler	Teisendaja – programm, mis konverteerib ühe kõrgema taseme keele sisendkood teise kõrgema taseme keele väljundkoodiks
TDD	*Test Drive Development* – testipõhine tarkvaraarenduses Lähenemisviis
TOPT	*Time-based one-time password* – ühekordne parool, mis kehtib ainult ühe tehingu või ühe kasutamisseansi kestel
UI	*User Interface* – kasutajaliides
URL	*Uniform Resource Locator* – veebiaadress
Kasutajalugu	Ingl. k. User story; kirjeldab süsteemis aktori poolt läbiviidavaid tegevusi
UUID	*Universally Unique Identifier* – universaalselt ainulaadne identifikaator
UML	*Unified Modeling Language* – üldotstarbeline noteeringukeel tarkvara, peamiselt suurte objektorienteeritud projektide spetsifitseerimiseks ja visualiseerimiseks
UX	*User Experience* – kasutajakogemus
XML	*Extensible Markup Language* – dokumentide märgendamiskeel, mis on nii masin- kui ka inimloetav
YAML	*YAML Ain't Markup Language* – inimsõbralik andmete serialiseerimise standard

Sisukord

Jooniste loetelu

Tabelite loetelu

Koodinäidete loetelu

1 Sissejuhatus

Meditsiiniseadmete tootmise valdkond on nii Euroopa Liidus kui ka globaalsest reguleeritud, et neis pakutavad tooted ja teenused oleksid lõppkasutajatele ohutud. Määrustega on kirjeldatud üldised nõuded millede saavutamiseks järgitakse rahvusvahelisi standardeid ning juhiseid. Tootekvaliteedi tagamiseks ja riskide maandamiseks kasutavad tootjad rahvusvahelisi ISO standardeid, millede nõuetele vastates peetakse meditsiiniseadmete tootjaid ka määrustega vastavuses olevateks.

Tulenevalt meditsiiniseadmete tootmise määrusest peavad ettevõtete kvaliteedijuhtimissüsteemis dokumendid olema kinnitatud ja kooskõlastatud allkirjadega. Meditsiiniseadmete tootmisvaldkonna kvaliteedijuhtimissüsteemi rahvusvahelise standardi nõuete puhul piisab dokumendi allkirjastamiseks lihtsast elektroonilisest allkirjast. Sellisel kujul antud elektrooniline allkiri ei taga allkirja usaldusväärsust ja terviklikkust ning ei ole seetõttu juriidiliselt samaväärne Eestis laialt levinud digitaalse allkirjaga.

Käesolev lõputöö analüüsibki meditsiiniseadmete tootmisvaldkonna kvaliteedijuhtimissüsteemides dokumentide elektroonilise allkirjastamise teemat ning selle väljakutseid Eesti kontekstis. Lõputöö uurib olemasolevaid tarkvaralisi kvaliteedi- ja dokumendihaldussüsteeme, mis on kättesaadavad ja sobilikud Eestis tegutsevatele väikse ja keskmise suurusega meditsiini- ja tervishoiuseadmete tootjatele.

Lõputöö eesmärgiks on pakkuda välja lahendus, mille abil on võimalik meditsiiniseadmete tootmisvaldkonna kvaliteedijuhtimissüsteemides dokumente digitaalseid allkirjastada, muutes need dokumendid Eestis juriidiliselt kehtivateks. Lahendus koosneb pilootprojektina loodavast API-rakendusliidesest, mille abil on võimalik liidestada Eestis laiemalt levinud kvaliteedijuhtimissüsteeme digitaalse allkirjastamislahendusega. Loodav pilootprojekt, ehk POC, võimaldab testida tarkvara toestatavust ja sobivust ning annab aluse jätkutegevuste planeerimiseks.

Lõputöö autor on tööalaselt seotud erinevate reguleeritud valdkondade kvaliteedi- ning dokumendihaldussüsteemidega pakkudes neid globaalsetele ettevõtetele, mis aitavad neil vastata oma eriala määrustest tulenevatele nõuetele. Autor on veendunud, et antud probleemi parimaks lahendamiseks Eestis on API-rakendusliidese arendamine, mis võimaldaks tagada vastavust kõige kasumlikul viisil.

1.1 Ülevaade taustast

Euroopa Liidu regulatsioonid meditsiiniseadmete tootmisvaldkonnas on kirjeldatud riigi või valdkonna regulatiivsete asutuse määrustes. Alates 26. mai 2021 kehtib Euroopa Liidus meditsiiniseadmete tootjate määrus MDR[1] 2017/745, mille nõuetele peavad vastama kõik Euroopa Liidu turul tegutsevad meditsiiniseadmete tootjad [1]. MDR 2017/745 määrusega on sätestatud turustavale tootele CE-märgise[2] vastavuskohustus, mis kinnitab toote vastavust määrusega. CE-märgise nõuetes on kirjeldatud, milliseid rahvusvahelisi ISO standardeid võivad tootjad rakendada vastamaks meditsiiniseadmete tootmise regulatiivsetele nõuetele kvaliteedijuhtimissüsteemide valdkonnas. Euroopa Liidus järgitakse valdavalt meditsiiniseadmete tootmisvaldkonnas standardi ISO 13485 kirjeldatud nõudeid ja juhiseid, mis aitab ettevõttel tagada tootekvaliteedi, hinnata ja maandada toote riske ning järgida meditsiiniseadmetarkvara arendamise parimaid tavasid [2].

Meditsiiniseadmete tootmise valdkonnas tuleks kindlasti lisaks ära mainida USAs FDA[3] poolt välja antud määrused. Globaalselt järgitakse meditsiiniseadmete tootmisvaldkonnas standardite ISO 13485, ISO 14971 ja IEC 62304 kirjeldatud nõudeid ja juhiseid [3]. Üldlevinud standardite nagu ISO 9001 and ISO 14001 järgimine võimaldab tootmis- ja tööstuse ettevõtetel taotleda neile vastavaid sertifikaate [4].

[1] MDR (*Medical Device Regulation*) – meditsiiniseadme tootjate regulatsioon

[2] CE-märgis – sertifitseerimistähis, mis kinnitab, et toodet on hinnatud vastavalt õigusaktidele [8]

[3] FDA (*Food and Drug Administration*) – USA Toidu- ja Ravimiamet [9]

Meditsiiniseadmete tootmisvaldkonnas peavad tootja igapäevased töökorraldused ja protseduurid olema kooskõlas ja vastavuses määratud rahvusvahelise ISO 13485 nõuetega, mille tulemusel loetakse neid ka vastavuses olevateks kvaliteedijuhtimissüsteemi puudutavate nõuetega määruses. Enamasti eeldab see organisatsioonilt kvaliteedijuhtimissüsteemi, ehk QMS'i ja dokumentide haldussüsteemi, ehk DMS'i juurutamist nii organisatsiooni- kui tootmisprotsesside haldamiseks, dokumenteerimiseks kui ka kooskõlastamiseks

1.2 Metoodika

Käesoleva lõputöö käigus selgitatakse esmalt lahti eksisteerivat probleemi, eksisteerivaid lahendusi ja nende puudusi ning pakutakse välja probleemile sobiv IT lahendus, mis jääks lõputöö skoopi, kuid võimaldaks edasi arendamist tulevikus. Lisaks käsitletakse ka loodud lahenduse võimalusi enda üleval pidamise kulude katmiseks.

Lahenduse analüüsi käigus tuuakse välja erinevad kasutajagrupid ning neid puudutavad nõuded kasutajaloode kujul, mis on grupeeritud funktsionaalseteks ja mittefunktsionaalseteks nõueteks. Samuti analüüsitakse erinevaid tehnilisi lahendusi ning põhjendatakse tagarakenduste ja kliendipoolseid eesrakenduste tehnoloogiate valikuid kui ka arendus- ning haldusvahendite valikuid.

Lahenduse valmimist on kirjeldatud erinevate osadena, mis käsitlevad eesrakenduse nõudeid kasutajaliidesele, andmebaasi disaini ning serveripoolse lahenduse valmimist. Lahenduse käigu lõpuosas on kirjeldatud ka lahenduse testimist ning edasiseid samme väljaspool lõputöö skoopi

2 Ülevaade probleemist

Meditsiiniseadmete tootmisvaldkonna kvaliteedijuhtimissüsteemides on oluline osa kontrollitud dokumentide haldamine. Standardis tulenevalt on meditsiiniseadmete tootjad kohustatud enda tootmisprotsessis järgima konkreetseid protseduure dokumentide dokumenteerimise, kinnitamise ja ajakohasena tagamise osas. Kvaliteedijuhtimissüsteemi oluline osa on seega ettevõttesisene dokumentide allkirjastamine nende kinnitamiseks ja kooskõlastamiseks.

Paberdokumentide allkirjastamisel on juriidiliselt kehtiv omakäeline allkirjastamine. Niipea kui tootjad loovad paberdokumentide asemel elektroonilisi dokumente, peavad nad käsitsi kirjutatud allkirjad asendama elektrooniliste või digitaalallkirjadega. Tänapäeval on elektroonilised dokumendid valdavalt asendanud paberdokumendid ning sel põhjusel on määruseid ja rahvusvahelisi standardeid aastate jooksul kaasajastatud ning sinna on lisatud nõudeid dokumentide elektrooniliseks allkirjastamiseks [5].

Euroopa Liidu meditsiiniseadmete tootjatele suunatud määruses on lubatud kasutada elektroonilist allkirjastamist omakäelise allkirjastamise asemel, kuigi määruses ei ole dokumendi elektroonilise allkirjastamise nõudeid selgelt reguleeritud ega kirjeldatud. Seetõttu on kasutuselevõetud erinevaid identifitseerimisevahendeid elektroonilise allkirjastamise tarbeks, näiteks allkirjastaja poolt PIN-koodi või salasõna sisestamist kui ka TOPT[1] salavõtme algoritmil põhinevaid generaatoreid. Sellisel kujul antud elektroonilised allkirju ei saa kontrollida ega tuvastada ning seetõttu vastavad need Euroopa Liidus ainult kõige madalamale elektrooniliste allkirjastamise tasemele. Vastavalt Euroopa Liidus kehtivale e-identimise ja e-tehingute määrusele eIDAS 910/2014 kvalifitseerub selline allkiri kui *lihtne elektrooniline allkiri (SES)* [6].

[1] TOPT (*Time-based one-time password*) – ühekordne parool, mis kehtib ainult ühe tehingu või ühe kasutamisseansi kestel [10]

Euroopa Liidus kehtivad elektrooniliste allkirjade tasemed vastavalt eIDAS 910/2014 määrusele on täpsemalt kirjeldatud peatükis „2.2.1 Elektrooniliste allkirjade tasemed Euroopa Liidus“.

Meditsiiniseadmete tootmise kvaliteedijuhtimissüsteemi rahvusvahelises standardis ISO 13485 kirjeldatud elektrooniline allkiri vastab Euroopa Liidus ainult tasemele *lihtne elektrooniline allkiri,* ehk SES. Selline kujul sooritatud elektrooniline allkiri ei vasta aga Euroopa Liidus kvalifitseeritud digitaalse allkirja nõuetele ning sel põhjusel ei ole juriidiliselt võrdsustatud omakäelise allkirjaga. Sel viisil elektrooniliselt allkirjastatud dokumentidel tekib küsitavus nende usaldusväärsuse, autentsuse ja terviklikkuse garanteerimise osas.

Kuigi meditsiiniseadmete tootjatele on saadaval laias valikus digitaalseid kvaliteedijuhtimise ja dokumentide haldussüsteemi, mis võimaldavad dokumentidele lisada elektroonilist allkirju, siis neis pakutav elektrooniline allkiri ei ole seega Euroopa Liidus ning Eesti kontekstis juriidiliselt kehtiv. Eestis on digitaalne allkirjastamine muutunud normiks ning enamik Eesti kodanikke eelistab tänapäeval juriidiliste toimingute sooritamiseks kasutada pigem elektroonilisi vahendeid. Eesti kodanikel ja e-residentidel on võimalus kasutada digitaalallkirja andmiseks mitmeid viise ning neist neli populaarsemat on: ID-kaart, Digi-ID kaart, Mobiil-ID ning Smart-ID.

Eestis on meditsiiniseadmete tootjatele probleemiks kvaliteedijuhtimise ja dokumentide haldussüsteemides kvalifitseeritud digitaalse allkirja võimaluste puudumine. Eestis on laialt levinud mitmeid viise dokumentide digitaalseks allkirjastamiseks, mida aga ei ole võimalik rakendata kvaliteedijuhtimise ja dokumentide haldussüsteemides.

Käesoleva lõputöö eesmärk on luua lahendus, mis võimaldaks Eesti meditsiiniseadmete tootjatel rakendada digitaalse allkirjastamise võimalusi oma kvaliteedi- ja dokumendihaldussüsteemides. Käesoleva lõputöös keskendutakse ainult väiksematele ja keskmise suurusega meditsiini- ja tervishoiuseadmete tootjatele, kes tegutsevad Eestis.

2.1 Eksisteerivad lahendused

Saadaval on laias valikus kvaliteedijuhtimise ja dokumentide haldussüsteemi, mis vastavad rahvusvahelistele ISO standarditele ja meditsiiniseadmete tootmisvaldkonna määrustele. Lõputöö kontekstis käsitletav valdkond on väga lai ning seetõttu on valikus saadaval alates lihtsatest lahendustest, mis on eeskätt loodud väikestele ettevõtetele kuni suurte ja keerukate süsteemideni, mis on suunatud suurtele rahvusvahelistele organisatsioonidele. Käesolevas lõputöös keskendutakse ainult väikse ja keskmise suurusega meditsiini- ja tervishoiuseadmete tootjatele, kes tegutsevad Eestis.

2.1.1 Lahenduste võrdluse lähtetingimused

Võrdlusse on valitud kvaliteedijuhtimise ja dokumentide haldussüsteemid, mis võimaldavad dokumente elektrooniliselt allkirjastada. Lähemalt uuritakse, millisele standardile elektrooniline allkiri vastab ning kas eIDAS (910/2014) määruse järgi vastab allkiri Eesti mõistes kvalifitseeritud digiallkirjaks.

Samuti on arvestatud toodete kättesaadavust, eelkõige lähtuvalt toodehinnast. Lõputöös käsitletakse eeskätt Eestis tegutsevad väikse ja keskmise suurusega meditsiiniseadmete tootjaid ning valdkonnas alustavaid iduettevõtteid. Selliste ettevõtete eelarved võivad tihtipeale olla piiratud mistõttu nad ei pruugi kasutada kallimaid ja keerukamaid dokumendihaldus lahendusi. Võrdlusse on ainult valitud ainult pilvepõhised *SaaS* lahendused ning välja on jäetud *On-Premises*[1] tarkvaratooted, mis eeldavad kallist riistvaralist võimekust.

2.1.1 Võrdlustabel

Järgnevalt on võrreldud eksisteerivaid lahendusi elektroonilise allkirjastamise ja kättesaadavuse alusel. Valimise valiti ainult lahendused, mis on Eesti turul olevate meditsiiniseadmete tootjatele hinnalt kättesaadav ning Eestis levinud. Võrdlustabel (Tabel 12) on suure mahukuse tõttu leitav Lisast 2.

[1] *On-Premises software* – tarkvara, mis nõuab käitlemiseks ettevõttelt füüsilist riistvara

2.2 Elektrooniline allkirjastamine

Eestis tähistab mõiste „digitaalne allkiri“ (ehk digiallkiri, digiallkirjastamine) ainult sellist signeerimisviisi, mis on seaduslikult kehtiv ning juriidiliselt võrdne omakäelise allkirjaga. Tuvastatud on, kes allkirja andis, ning kindlustatud, et keegi kolmas ei ole allkirjastatavat dokumenti peale selle allkirjastamist muutnud. Mõiste „elektrooniline allkiri“ (ehk e-allkiri) on aga laiem ja hõlmab erinevaid allkirjatasemeid – seal hulgas ka digiallkirju. Järgnevas lõigus on välja toodud erinevad elektrooniliste allkirjade tasemeid Euroopa Liidus.

2.2.1 Elektrooniliste allkirjade tasemed Euroopa Liidus

Alates juulist 2016 hakkas kehtima Euroopa Liidu otsekohalduv rakendusakt, mis jõustas e-identimise ja e-tehingute määruse, ehk eIDAS 910/2014. [7] Sellest tulenevalt eristatakse neljal tasemel e-allkirju [8]:

1. **tase – QES** (Qualified Electronic Signature) . Kvalifitseeritud digitaalse allkirja kõrgeim tase, mis on võrdsustatud omakäelise allkirjaga ja mida nimetatakse Eestis ka digiallkirjaks. Allkiri vastab standardites kehtestatud tehnilistele nõuetele. Kontrollitud on nii allkirja omaniku taust kui ka sertifikaadi väljaandja taust.
2. **tase – AdES/QC** – Täiustatud (Advanced) digitaalne allkiri koos kvalifitseeritud sertifikaadiga (Qualified Certificate). Allkiri vastab standardites kehtestatud tehnilistele nõuetele. Kontrollitud on nii allkirja omaniku taust kui ka sertifikaadi väljaandja taust.
3. **tase – AdES** (Advanced Electronic Signature) – Allkiri vastab standardites kehtestatud tehnilistele nõuetele, aga allkirja andmiseks kasutatud sertifikaadi omaniku taust ja sertifikaadi väljaandja taust ei pruugi olla teada.
4. **tase – Muud elektroonilised allkirjad ehk lihtne elektrooniline allkiri (SES – Simple Electronic Signature)** – igasugused muud elektroonilisel kujul antud allkirjad, mis ei vasta kehtivatele standarditele.

2.2.2 Elektrooniliste allkirjastamise lahenduste võrdlus

Järgnevalt on uuritud ja võrreldud erinevaid allkirjastamise lahendusi, mis võimaldavad dokumente elektrooniliselt allkirjastada. Taaskord on lähemalt uuritud vastavust

Euroopa Liidu määrusega eIDAS 910/2014. Eesti kontekstis lähtuvalt on samuti uuritud, kas võrreldavas lahenduses on Eesti kodanikel või e-residentidel võimalus kasutada mugavalt dokumentide digitaalseks allkirjastamiseks ID-kaardi, Digi-ID kaardi, Mobiil-ID ning Smart-ID toetavaid lahendusi. Valimisse valiti järgmised lahendused, mida võrreldi eelnevalt kirjeldatud lähtetingimuste alustel.

- *DigiDoc4 klient* – kasutaja arvutisse paigaldatav tarkvara, mille abil saab kiiresti, lihtsalt ja mugavalt Eestis seaduslikult siduvaid dokumente digiallkirjastada ja digiallkirjade kehtivust kontrollida; faile avada, salvestada ja jagada [9]. Tarkvara ei ole võimalik liidestada eelnevalt valitud kvaliteedijuhtimise ja dokumentide haldussüsteemidega. Võimalik kasutada dokumentide allkirjastamisel ID-kaardi, Digi-ID kaardi, Mobiil-ID'd ning Smart-ID'd
- *Dokobit* – Eestis loodud dokumentide allkirjastamiseks veebipõhine teenus, mis võimaldab dokumente allkirjastada kvalifitseeritud digiallkirjaga [10]. Võimalik on kasutada dokumentide allkirjastamisel ID-kaardi, Digi-ID kaardi, Mobiil-ID'd ning Smart-ID'd. Teenust on võimalik integreerida teiste süsteemidega kasutades kas kohendatud kasutajaliidest, ehk *Documents Gateway* lahendust või ainult API põhiseid lahendusi [11].
- *Adobe Sign* – Adobe poolt loodud pilvepõhine elektroonilise allkirjateenus, mis võimaldab kasutajal brauseri või mobiilseadme abil allkirjaprotsesse saata, allkirjastada, jälgida ja hallata [12]. On üks osa Adobe Document Cloud'i teenuste komplektist. Adobe Sign digitaalne allkirjastamine vastab eIDAS määruse alusel 1. taseme kvalifitseeritud digiallkirjaks (QES) ja 2. taseme täiustatud digiallkirjaks (AdES/QC). [13] Teenuses ei ole võimalik kasutada dokumentide allkirjastamisel ID-kaardi, Digi-ID kaardi, Mobiil-ID või Smart-ID lahendusi.

2.3 GDRP ja HIPAA

Meditsiiniseadme tootjad võivad käidelda organisatsiooni siseses kvaliteedijuhtimise ja dokumentide haldussüsteemis tundlikke isikuandmeid, näiteks andmeid patsiendi terviseteabe kohta [14]. Andmeid töötlev ettevõtte või organisatsiooni on peamine andmetöötleja, ehk „Vastutav töötleja", kes vastutab isikuandmete töötlemise eest ning neile kohaldatakse andmekaitse määrusi [15]. Meditsiiniseadme tootjale pilvepõhiseid teenust pakkuv ettevõte, kes säilitab ja töötleb andmeid „Vastutav töötleja" volitustel, ehk

„Volitatud töötleja“, peab samuti järgima andmekaitse määrusi ning tellijaga sõlmitud kohustusi seoses andmekaitse määrusega. Seetõttu on oluline, et loodav lahendus oleks vastavuses kohaldavate andmekaitse määrustega ning lahenduse loomisel tuleb andmekaitse nõuetega ka arvestada. Järgnevatel tutvustakse määrusi, mida kohaldatakse meditsiiniseadmete tootjatele ja nende partneritele.

2.3.1 GDPR

GDPR[1], ehk Euroopa Liidu isikuandmete kaitse üldmäärus, asendas senise Euroopa andmekaitsedirektiivi (*Data Protection Directive*), mis kehtis alates 1995. aastast, kuna selle loomise ajal ei olnud Internet nii levinud ja antud direktiivis ei ole reguleeritud praegu kasutatavad andmete kogumisviise [16].

GDPR'i nõuete täitmiseks tuleb muuta ettevõttesiseseid protseduure ja tehnilisi lahendusi andmete käitlemise kohta. Iga tegutsev ettevõte või asutus kogub füüsiliste isikute kohta andmeid ja seetõttu peab ettevõte eristama, millised õigused on neil määruse raames seoses andmete töötlemisega ja millised kohustused on andmete turvalisuse tagamisel. Nõuete rikkumisel on maksimaalne trahvisumma 20 miljonit eurot või 4% ettevõtte ülemaailmsest käibest, olenedes sellest, kumb on suurem [17].

Lisaks eristab seadus vastutavat ja volitatud isikuandmete töötlejat ning ettevõte peab määratlema, kumb osapool ta on. Vastutav töötleja määrab kindlaks isikuandmete töötlemise eesmärgid ja selleks kasutatavad vahendid. Volitatud töötleja töötleb isikuandmeid ainult vastutava töötleja nimel ja tema näol on enamasti tegemist ettevõttevälise kolmanda isikuga. Volitatud töötleja kohustused vastutava töötleja ees tuleb kindlaks määrata lepinguga või mõne muu õigusakti alusel, sealhulgas tuleb sätestada kord, mida tehakse isikuandmetega pärast lepingu lõpetamist [18].

2.3.2 HIPAA

HIPAA[2] on alates aastast 1996 rakendatud USA seadus, mis piirab kaitstud terviseteabe kasutamist tervishoiuorganisatsioonide poolt [19]. Kõik tervishoiuorganisatsioonid või

[1] *General Data Protection Regulation*

[2] *Health Insurance Portability and Accountability Act*

nendega seotud äripartnerid on seadusega kohustatud järgima HIPAA-standardit. Meditsiiniseadme tootja, kes soovivad tegutseda või siseneda USA turule, peavad end vastavusse viima HIPAA-standardiga.

HIPAA ja GDPR on juhtivad raamistikud, mis kaitsevad üksikisikute privaatsust [19]. Loodava tarkvara loomisel on mõistlik vältida isikuandmete ja terviseteabe töötlemine või vajadusel teha seda minimaalselt ning nii vähe kui võimalik. Kui ole teada, milliseid andmeid kliendid lahenduses hoidma hakkavad, siis on alati hea eeldada, et ollakse volitatud töötleja vastavalt GDPR'le. Samuti on oluline jälgida, et loodavas lahenduses kasutatavad kolmanda osapoole pilveteenused oleksid vastavuses GRPR ja HIPAA nõuetega.

2.4 Uue lahenduse skoop

Eksisteerivate lahenduste puuduseks Eesti kontekstis on võimaluste puudumine dokumente digitaalselt allkirjastada juriidiliselt kehtiva kvalifitseeritud digitaalse allkirjaga. Eesti kodanikel või e–residenditel on võimalus kasutada mugavalt dokumentide digitaalseks allkirjastamiseks ID-kaardi, Digi-ID kaardi, Mobiil-ID ning Smart-ID toetavaid lahendusi, kuid neid lahendusi ei ole võimalik kasutada eelnevalt võrreldud tarkvaradel ja platvormidel.

Probleemi lahendamiseks pakub lõputöö autor välja 2 potentsiaalset lahendust. Esimeseks võimalikuks viisiks probleemi lahendamiseks on arendata uus kvaliteedijuhtimise ja dokumentide haldussüsteem, mis võimaldaks dokumente allkirjastada Eestis ja Euroopa Liidus kehtiva kvalifitseeritud digitaalse allkirjaga. Pakutud lahenduse realiseerimisel on kaks puudust:

- *Kulukas arendus* – arvestades konkureerivate lahenduste keerukust ja pakutavaid võimalusi ning rahvusvahelisest standardist ISO 13485 tulevaid nõudeid arendatavale kvaliteedijuhtimissüsteemile, siis võib eeldata lahenduse arendusele suurt mahtu ning kulukat investeeringu vajadust.
- *Piiratud tulu* – Eesti siseturu väiksus seab kindlad piirid potentsiaalsele sihtgrupile ja tuluteenimisele ning seetõttu on raske luua konkurentsi pakkuvad tooted.

Teine võimalik lahendus on laiendada juba eksisteerivat kvaliteedijuhtimise ja dokumentide haldussüsteemi. Mitmed eksisteerivad lahendused võimaldavad tarkvaraplatvormile lisada kolmanda osapoole tooteid, ehk *plugin*'eid[1]. Lisatava *plugin*'i kaudu on võimalik tuua tarkvaraplatvormile vajalikud funktsionaalsused dokumentide digitaalseks allkirjastamiseks. *Plugin*'i arendamine võimaldab ainult keskenduda funktsionaalsustele, mis on vajalikud dokumentide digiallkirjastamiseks. Tarkvaraplatvormi valikul tuleks arvestada järgneva kolme teguriga:

- *Populaarne* – eksisteeriva tarkvaraplatvormi valikul tuleks arvestada, et valitav peab Eestis kontekstis juba olema piisavalt laialdaselt kasutusel ning eeskätt meditsiiniseadmete tootjate sihtgrupis.
- *Laiendatav* – valitav tarkvaraplatvorm peab tehniliselt võimaldama *plugin*'ite lisamist, mille kaudu lisada funktsionaalsust dokumentide allkirjastamiseks.
- *Kättesaadavus* – valivat tarkvaraplatvorm peab olema sihtgrupis hinnalt kättesaadav. See hõlmab ka rakenduse käitlemise kulutusi ning seetõttu eelistavalt valitav lahendus peab olla pilvepõhine *SaaS* äritarkvara, mis ei nõua riistvarale ja kompetentsile lisakulusid.

Võttes arvesse eelnevalt loetletud võimalikke variante lõputöö kontekstis oleva probleemi lahenduseks ning eksisteerivate lahenduste võrdlustabelit, pakub käesoleva lõputöö autor välja, et probleemi lahendamiseks on parim viis kasutada juba eksisteerivat tarkvaraplatvormi Confluence Cloud ning laiendades tarkvaraplatvormi vajalikke funktsionaalsustega uue lisatava toote kaudu.

Tarkvaraplatvorm Confluence Cloud on nii globaalselt kui ka Eestis laialdaselt kasutusel ja kättesaadav kui kvaliteedijuhtimise ja dokumentide haldussüsteem. Confluence Cloud'il on paindlik hinnastamise mudel lähtuvalt kasutajate arvust ning *SaaS* lahendusena puuduvad käitlemisel kõrvalkulud. Tarkvaraplatvormile on mugav arendata kolmanda osapoole tooteid *plugin*'ite kaudu, mis pakuvad rohkeid võimalusi tarkvaraplatvormi laiendamiseks ning integreerimiseks teiste süsteemidega.

[1] *Plugin* - lisandprogramm, mis on hõlpsalt installitav ja laiendab ning täiendab senise programmi võimalusi

Confluence Cloud võimaldab ehitada *plugin*'ite abil ainult piiratud võimalustega programme, mis on eeskätt mõeldud kasutajaliidese laiendamiseks. Keerukamate funktsionaalsuste lisamiseks tarkvaraplatvormile on vajalik luua eraldiseisev serveripoolne teenuskiht, ehk tagarakendus (*backend*), mida on võimalik liidestada plugin'i kasutajaliidesega, ehk eesrakendusega (*frontend*).

Käesoleva lõputöö skoobis lootakse uus API-põhine tagarakendus, mis pakub teenuseid dokumentide tõendamiseks digitaalse allkirjastamise kaudu. Loodav API-põhine tagarakendus on eeskätt mõeldud liidestamiseks tarkvaraplatvormiga Confluence Cloud *plugin*'ite kaudu ning millega lisatakse tarkvaraplatvormile võimalused dokumentide allkirjastamiseks kvalifitseeritud digiallkirjaga.

Lõputöö skoobi ei mahu mitmete funktsionaalsuste arendamine nagu allkirjastavate komplektide loomine, töövoogude allkirjastamine, tegevuste ajalugu ning teavitused. Käesolev arenduses keskendub põhiliste funktsionaalsuste loomisele dokumentide kvalifitseeritud digitaalseks allkirjastamiseks. Samuti suure mahukuse tõttu jätab autor lõputöö skoobist välja tarkvaraplatvormil Confluence Cloud uue *plugin*'i arendamise, kuid siiski analüüsi käigus tehakse ära vajalik eeltöö ning seadistakse eesrakendus, et selgitada välja tagarakendusele esitatavaid eeldusi. Järgmises arendusetapis on planeeritud liidestada API-põhine rakendus uue loodava Confluence Cloud *plugin*'iga

Dokumentide allkirjastamiseks kasutatakse Dokobit'i teenust, mis pakub läbi integratsioonide mugavat ja turvalist viisi tarkvaraplatvormil Confluence Cloud dokumentide allkirjastamiseks. Dokobit teenused liidestatakse loodava API-põhise rakendusega ning järgmises skoobis võimaldab *plugin*'iga liidestamine lõppkasutajal sooritada digitaalset allkirjastamist tarkvaraplatvormi kasutajaliidesest lahkumata.

Tarkvaralatvorm Confluence Cloud on Eestis laialt kasutatav ja kättesaadav ning on parim võimalus jõuda sihtgrupini. Arenduses tuleb samuti arvestada pikema perspektiiviga, et tulevikus võib olla äriliselt vajalik liidestada API-rakendusliides ka teiste kvaliteedijuhtimise ja dokumentide haldussüsteemidega.

2.5 Uue lahenduse ärilise tasuvuse analüüs

Lõputöö kontekstis loodava toode eesmärk on olla äriliselt kasumlik. Ärilise tasuvuse saavutamiseks on minimaalne eesmärk teenida vähemalt nii palju tulu, et katta domeeni- ning serverikulusid ning Dokobit' teenuse kasutamise kulud. Domeeni- ja serverikulude suurusjärk on minimaalselt 50 eurot kuus arvestades antud lõputöös valitud domeeni- ning serveriteenuse pakkujaid, mis on täpsustatud hilisemates peatükkides. Dokobit'i allkirjastamise teenuse minimaalne kuutasu on 50 eurot, millele lisandub 0,36 igale allkirja kohta.

Loodav rakenduse luuakse mikroteenuste arhitektuuril, mis võimaldab serveris käideldavaid teenuseid orkestreerida paindlikult, võimaldades skaleeritavust vastavalt nõudlusele. Seetõttu saab prognoosida käitlemise kulude kasvu vastavalt kasutajate arvu kasvule. API-rakendusliidese arhitektuur on täpsustatud hilisemates peatükkides. Samuti sõltub Dokobit'i allkirjastamise teenuse kulud vastavalt kasutajate arvust. Seetõttu on mõistlik kasutada hinnastamisel kasutajate arvu põhist mudelit.

Tarkvaraplatvormiga Confluence Cloud kaasneb kolmanda osapoole toodete müüki võimaldav *Atlassian Confluence Marketplace*, mis on kättesaadav kõigile kasutajatele [20]. Tulu teenimiseks on parim võimalus lisada lõputöö kontekstis loodav toode tarkvaraplatvormi rakenduste poodi, mis pakub järgmisi eeliseid:

- Levitamine ja turustamine – Atlassian'i ökosüsteemil on globaalselt suur kasutajate arv üle 10 miljoni kasutajaga ning 200 000 maksva kliendiga, kellel on lihtne ja mugav ligipääs loodavale tootele. Rakenduste poes *Confluence Marketplace* on lõputöö kontekstis müütavat toodet lihtne levitada ning turundust on võimalik teha ilma lisakuludeta.
- Lihtne toode proovimine ja ostmise – kliendid saavad rakendusi proovida või osta otse rakenduste poest. Kasutajatel on 30-päevane proovimise periood.
- Hinnastamine – vastavalt *Confluence Marketplace* rakendute poe hinnastamise poliitikale tasuvad kliendid rakenduste eest vastavalt kasutajate arvule tarkvaraplatvormil Confluence Cloud. Väljaandja peab hinnakujundamisel määrama rakenduse hinna ühe kasutaja põhiselt, mis lihtsustab hinnakujundamist.

- Tuluteenimine – toote müügist teenitud brutotulu jagatakse toote väljaandja ja Atlassian'i vahel vastavalt hinnastamise poliitikale. Atlassian maksab saadud brutotulust 85 protsendi toote väljaandjale.
- Soodustused – Rakenduste pood võimaldab hallata sooduskoode ning toote väljaandjal on võimalik neid kasutada turunduses. Samuti on võimalik luua personaalseid sooduskoode.
- Maksmine – Atlassian pakub toode eest tasumist läbi enda maksesüsteemi. Maksmisega seotud protsesside eest hoolitseb Atlassian ning igakuiselt makstakse tulu toote väljaandjale. Võimalik luua ülevaatlikke raporteid.
- Väiksemad omakulud – Atlassian hoolitseb toode levitamise, käitlemise ja tasumise protsesside eest, mis vähendab investeeringute vajadust ning see läbi aitab vähendata toode omahinda.

API-põhine rakenduse minimaalsed domeeni- ning serveriteenuste kulud on 50 eurot kuus, millele lisandub vähemalt 50 eurot koos Dokobot'i teenuse kuutasuna. Võrreldes paljude rakenduste hindu rakenduste poes *Confluence Marketplace*, siis lõputöö kontekstis loodava lahenduse eeldavatav hind kasutaja koht on 3 eurot kuus. Võttes arvesse, et rakenduste pood maksab toote omanikule välja 85 protsendi brutotulust, siis on kulude katteks vajalik, et *Confluence Marketplace* rakenduste poes oleks vähemalt 40 maksvat kasutajat[1] [20].

Arvestades meditsiiniseadmete tootjate suurt tegevusvaldkonda, siis on lõputöö kontekstis loodaval tootel potentsiaali olla tulevikus äriliselt kasumlik. Tulevikus on mõeldud loodavat API-põhist rakendust ka liidestada teiste platvormide ja tarkvaradega, mis veelgi laiendab tulude teenimise võimalust.

[1] Arvutuskäik – 40 (kasutajat) * 3 (rakenduse hind kasutaja kohta, eurot) * 0,85 (väljamakstav protsent)

3 Loodava lahenduse analüüs

Käesolevas peatükis kirjeldatakse ära loodava lahenduse funktsionaalsused, mis selgitatakse välja funktsionaalsete ja mittefunktsionaalsete nõuete kaudu. Peatüki teises osas kirjeldatakse detailselt eesmärgi saavutamiseks kasutatavaid tehnoloogiad ning põhjendatakse vastavate valikute tegemist.

3.1 Nõuete määramine

Nõuete analüüsi eesmärk on selgitada välja lõputöö kontekstis loodava lahenduse esitatavad ärinõuded ning lõppkasutajate ootused süsteemi funktsionaalsuse osas. Analüüsi käigus selguvad nõudeid on aluseks kasutajalugude koostamisel, mille põhjal kirjeldatakse tulevikus loodava kasutajaliidese

Lõputöö kontekstis loodavale lahenduse probleemi analüüsi etapis on selgunud järgmised lähtetingimused, millega tuleb arvestada nõuete selgitamisel:

- Lahendus peab vastama Euroopa Liidus meditsiiniseadmete tootjatele määrusele 2017/745 MDR.
- Meditsiiniseadmete tootmisvaldkonna kvaliteedijuhtimissüsteemi peab vastama rahvusvahelise standardile ISO 13485.
- Elektroonilise allkirjastamise nõuded ning tasemed on määratud Euroopa Liidu määruses 910/2014 eIDAS.
- Loodav rakendus peab arvestama GDPR ja HIPAA andmekaitse seadustega ning seetõttu tuleks vältida isikuandmete ja terviseteabe talletamist või vajadusel teha seda minimaalselt ning nii vähe kui võimalik.
- Tuleks eeldada, et tarkvara arendajana ollakse volitatud töötleja vastavalt GDPR'le.
- Lahendus realiseeritakse tarkvaraplatvormil Confluence Cloud ning loodav API-rakendusliides liidestakse *plugin*'i kaudu.

- Lahenduse kasutajad on autenditud [1]Atlassiani kasutajakontoga[2] kasutajad ning tarkvaraplatvormil Confluence Cloud tegevustele kasutajaõigused määratakse autoriseerimise[3] protsessi kaudu.

Nõuete määramisel on kasutatud lähtetingimustes loetletud määruste ja standardite materjale ja dokumentatsioone.

Funktsionaalsete ning mittefunktsionaalsed nõuete kaardistamisel ja tarkvaranõuete klassifitseerimisel on kasutatud kahte erinevad analüüsi mudelit ja metoodikat. Järgnevalt on lõputöös kirjeldatud MoSCoW ja FURPS nõuete analüüsi meetodeid ning esitatakse nende põhjal läbiviidud analüüsi tulemeid.

Peatükki lõpus on käsitletud nõudeid, mida soovitakse edasi arendada peale praeguse skoobi valmimist.

3.1.1 MoSCoW meetod

MoSCoW meetod on prioritiseerimistehnika, mille abil saab hinnata nõuete olulisust. MoSCoW analüüs meetodi järgimine aitab väldita üleliigse arendamisest. MoSCow meetodi loojaks on Dai Clegg, kes tutvustas seda esmakordselt 1994 aastal oma raamatus „*Case Method Fast-Track: A Rad Approach*“. [21] MoSCoW on tuletatud ingliskeelsetest *must* ehk peab, *should* ehk peaks, *could* ehk võiks ja *won't* ehk ei tee, mis kirjeldab erinevaid prioritiseerimise klasse. [22] See tehnika aitab mõista, mida tegelikult on vaja, et väärtust luua ja mis on kena, kui oleks.

MoSCoW meetodi prioriteetide klassifikatsioon on toodud järgnevas tabelis (Tabel 1):

[1] Autentimine on protsess, millega üks kasutaja, süsteem või muu olem (objekt) saab kontrollida teise olemi väidetava identiteedi tõesust, tavaliselt mingit tüüpi identsustõendi alusel.

[2] Kasutajakonto on kasutaja ning infoteenuse vahele kehtestatud seos; kasutajakontole kinnistatakse kasutajanimi ja parool; kasutajakontod võivad olla jaotatud erinevate õigustega tasemetesse.

[3] Autoriseerimine ehk volitamine on protsess, mis annab (või keelab) õiguse ligi pääseda ressurssidele.

Tabel 1. Prioriteetide klassifikatsioon MoSCoW meetodi alusel.

Klassifikatsioon	**Tähendus**
„Peab olema“ ehk *must have*	Lisatakse projekti. Ilma nendeta ei ole mõtet projekti realiseerida. Funktsionaalsused, mis julgustavad kasutajat seda tarkvara kasutama.
„Peaks olema“ ehk *should have*	Praegune projektiplaan sisaldab. Kuuluvad realiseerimisele. Kui tekib vajadus, siis võidakse välja jätta. Funktsionaalsused, mis teevad tarkvara kasutamist meeldivamaks, kuid neid saab lisada ka hiljem.
„Võiks olla“ ehk *could have*	Praegune projektiplaan ei näe ette nende lisamist. Kui olukord nõuab, siis võidakse mõned projekti kaasata. Funktsionaalsused, mis ei juurde anna märkimisväärset ärilist väärtust.
„Ei pea olema“ ehk *won't have*	Ei kaasata realiseeritavasse projekti. Funktsionaalsused võivad olla näiteks kunagi tulevikus olulised, kuid hetkel on iteratsiooni mahust väljas

3.1.2 FURPS mudel

FURPS on tarkvaranõuete klassifitseerimise mudel, mis aitab hinnata tarkvara kvaliteeti. Mudel on loodud 1987 aastal Robert Grady ja Hewlett-Packardi poolt. [22] *IBM Rational Software* laiendas aastal 2000 FURPS mudelit ning uueks nimeks sai FURPS+. Mudeli järgi jagunevad kasutaja poolt esitatavad nõuded funktsionaalseteks ja mittefunktsionaalseteks nõueteks. [23]

Funktsionaalsed nõuded vastavad küsimusele, mida tarkvara peab tegema ning mittefunktsionaalsed, kuidas tarkvara peab vajalikke funktsioone täitma.

FURPSI järgi klassifitseeritakse tarkvara kvaliteediomadused on välja toodud järgnevas tabelis (Tabel 2):

Tabel 2. FURPS mudeli kvaliteedifaktorid ja kriteeriumid.

Klassifikatsioon	**Kriteeriumid**
Funktsionaalsus (*Functionallity*)	Võimalused, suutlikkus, turvalisus
Kasutatavus (*Usability*)	Järjepidevus, inimfaktor, kasutaja dokumentatsioon, õppematerjalid
Töökindlus (*Reliability*)	Käideldavus, vigade tihedus, taastuvus, prognoositavus
Jõudlus (*Performance*)	Reaktsiooniaeg, korrektsus, läbilaskevõime, taasteaeg, ressursside kasutatavus
Toetatavus (*Supportability*)	Adapteeritavus, hooldatavus, konfigureeritavus, testitavus, ühilduvus, laiendatavus

FURPS+ klassifitseerib nõuded lisaks eelnevale veel: [25]

- disainile – täpsustab või piirab võimalusi süsteemi kavandamisel;
- implementatsioonile – täpsustab või piirab süsteemi koodi või konstruktsiooni;
- liidestele – täpsustab, milliste väliste elementidega peab süsteem suhtlema, või suhtluses kasutatavate vormide või tegurite piirangud;
- infrastruktuurile – täpsustab majutuseks mõeldud riistvarale esitatud füüsilised piirangud;

3.1.3 Funktsionaalsed nõuded

Funktsionaalne nõue on avaldus selle kohta, kuidas süsteem peab käituma ning määrab mida süsteem peaks tegema, et vastata kasutaja vajadustele või ootustele. Funktsionaalsete nõuete määramisel on nõudeid prioritiseeritud MoSCoW analüüsi metoodika abil ning igat nõuet on hinnatud vastu kirjeldatud klassifikatsiooni.

Järgnevalt on kirjeldatud funktsionaalsete nõuete tulemid koos MoSCoW prioritiseerimisega. Suure mahukuse tõttu on tulemite tabel (Tabel 13) leitav Lisa 3 alt.

3.1.4 Mittefunktsionaalsed nõuded

Mittefunktsionaalsed nõuded defineerivad selle, kuidas süsteem peab vajalikke funktsioone täitma. Mittefunktsionaalseid nõudeid võib pidada ka piiranguteks või kvaliteedinõueteks.

Mittefunktsionaalsed nõuded jaotati FURPS mudeli järgi järgnevalt [22]:

- funktsionaalsus (*functionality*)
- kasutatavus (*usability*)
- töökindlus (*reliability*)
- jõudlus (*performance*)
- toetatavus (*supportability*)

Järgnevalt on kirjeldatud *funktsionaalsusega* seotud mittefunktsionaalsed nõuded, mis aitavad tagada rakenduse korrektse toimimise turvalises viisil. Mahukuse tõttu on tabel (Tabel 14) leitav Lisa 4 alt.

Järgnevalt on kirjeldatud *kasutatavusega* seotud mittefunktsionaalsed nõuded, mis aitavad tagada kasutajaliidese loomise tulemusena saavutataks kõige optimaalsem kasutajakogemus. Mahukuse tõttu on tabel (Tabel 15) leitav Lisa 5 alt.

Järgnevas tabelis (Tabel 3) on kirjeldatud *töökindlusega* seotud nõuded, mis määravad nõuded käideldavusele, vigade esinemisele, taastuvusele ja töökindluse prognoositavusele:

Tabel 3. Töökindluse mittefunktsionaalse nõuete kirjeldus.

Kood	**Töökindluse mittefunktsionaalsete nõuete kirjeldus**
MF-REL1	Süsteem peab olema kasutajatele kättesaadav 24/7
MF-REL2	andmeid tuleb varundada vähemalt üks kord päevas, maksimaalne andmekadu (RPO) võib olla 24 tunni andmed
MF-REL3	ühe katkestuse maksimaalne kestvus võib olla kaks tundi, kuu lõikes summeeritult kuus tundi.
MF-REL4	tehnilise rikke korral peab süsteem olema taastatav ühe tööpäeva jooksul.
MF-REL5	süsteemi tuleb arenduse faasis põhjalikult testida

Järgnevalt on kirjeldatud *jõudlusega* seotud mittefunktsionaalsed nõuded, mis määravad nõuded andmete kiirusele süsteemis, reaktsiooniaeg, taasteaeg ning käivitamisele kuluv aeg:

Tabel 4. Jõudluse mittefunktsionaalsete nõuete kirjeldus.

Kood	**Jõudluse mittefunktsionaalsete nõuete kirjeldus**
MF-PER1	süsteemi üldiseks reaktsiooniajaks on üks sekund
MF-PER2	süsteem peab võimaldama vähemalt 50 kasutaja paralleelset tööd ilma, et tekiks reaktsiooniajas olulist muutust.
MF-PER3	mitteaktiivse kasutaja sessioon lõpetatakse viie minuti pärast.
MF-PER4	süsteem peab taaskäivituma viie minutiga.
MF-REL5	süsteemi tuleb arenduse faasis põhjalikult testida

Järgnevalt on kirjeldatud *toetatavatega* seotud nõuded, mis määravad nõuded adapteeritavusele, hooldatavusele, konfigureeritavusele, testitavusele, ühilduvusele ja laiendatavusele. Mahukuse tõttu on tabel (Tabel 16) leitav Lisa 6 alt.

Lisaks FURPS mudeli mittefunktsionaalsed nõuete jaotusele lisati veel täiendav jaotus FURPS+ klassifikatsiooni alusel. FURPS+ klassifikatsiooni järgi lisada täiendavad nõuded *implementatsioonile*, mida on kirjeldatud järgnevas tabelis (Tabel 5):

Tabel 5. Implementatsiooni mittefunktsionaalsete nõuete kirjeldus.

Kood	**Implementatsiooni mittefunktsionaalsete nõuete kirjeldus**
MF-IMP1	Rakendus peab olema tehniliselt tükeldatud vastavalt loogilisele jaotusele ning olema eraldi versiooneeritavad ja paigaldatavad
MF-IMP2	Koodibaas peab olema mugavalt hallatav tsentraalses repositooriumis.
MF-IMP3	koodibaas peab olema kirjutatud rahvusvahelises keeles.
MF-IMP4	Kõik API-otspunktid peavad olema dokumenteeritud OpenAPI 2.0 standardi aluses
MF-IMP5	Infosüsteemides on eessüsteemid (front end; presentatsiooni kiht) ja tagasüsteemid (back end; äriloogika kiht) arhitektuuriliselt selgelt lahutatud ja eraldi paigaldatavad.
MF-IMP6	Koodibaas peab olema dokumenteetritud kasutades README.md faili

3.1.5 Kasutajatasemed ja aktorid

Loodav süsteem on mõeldud ainult organisatsooni sisemiseks kasutamiseks ning süsteemile ligipääs on vaid lubatud autentinud süsteemi kasutajatele. Sisse loginud[1] kasutajale on lubatud tegevuste sooritamine talle volitatud kasutajaõiguse taseme piires.

Süsteemis on neli kasutajaõiguste taset tulenevalt eelnevalt selgitatud nõuetest:

- *I tase* – lubatud ainult dokumentide otsimise, vaatamise ja eksportimise tegevused;
- *II tase* – lubatud ainult dokumentide otsimise ja vaatamise, allkirjastamise tegevused;
- *III tase* – lubatud dokumentide lisamise, muutmise, avaldamine, allkirjastajate määramise, allkirjastamise, otsimise, vaatamise, peitmise/näitamise tegevused;
- *IV tase* – lubatud dokumentide lisamise, muutmise, avaldamine, allkirjastajate määramise, allkirjastamise, otsimise, vaatamise, eksportimise, peitmise/näitamise ning kustutamise ja kasutajate halduse tegevused;

Funktsionaalsete nõuete põhjal määrati järgnevad aktorid[2], mis jaotati vastavalt tegevustele ja õigustele järgmiselt:

- *Tavakasutaja* – omab esimese taseme kasutajaõigusi. Selleks kasutajaks on näiteks tavatöötaja, kellel on ligipääs dokumentide otsimiseks ja vaatamiseks;
- *Dokumenti kinnitaja* – omab teise taseme kasutajaõigusi. Dokumenti kinnitaja on isik, kes kinnitab ning allkirjastab dokumente;
- *Dokumenti koostaja* – omab kolmanda taseme kasutajaõigusi. Selleks kasutajaks on spetsialist, kes koostab dokumente ning valmistab neid välja ette allkirjastamiseks;
- *Admistraator* – omab neljanda taseme kasutajaõigusi. Kasutaja ülesandeks on hallata süsteemi lisatud dokumente ja kasutada süsteemi poolt genereeritavaid väljundeid. Lisaks süsteemis kasutajate haldamine ning ligipääsude andmine;

[1] Sisselogimine on volitatud kasutajal kasutamisseanssi alustada võimaldav ja volitamatuid tõkestav identifitseerimise ja autentimise protseduur.

[2] Aktor – süsteemi kasutajat esindav roll, mis kirjeldab süsteemis tema poolt läbiviidavaid tegevusi

3.1.6 Kasutajalood

Kasutuslugude kirjeldamine (*Use Case*) on laialtlevinud praktika, mille abil saab kirjeldada lõppkasutaja poolt läbiviitavaid tegevusi, mis annavad kasutajale reaalset väärtust. Kasutuslugude kirjeldamiseks määratleti esmalt aktorid vastavalt süsteemi tegevustele ja õigustele. Järgnevalt on kirjeldatud iga aktori poolt tehtavad tegevused kasutajalugudena.

Tavakasutaja põhised kasutajalood on kirjeldatud järgnevas tabelis (Tabel 6):

Tabel 6. Tavakasutaja põhised kasutajalood.

Kood	**Tavakasutaja põhine kasutajalugu**	**Nõue**
UC-T1	Tavakasutajana soovin sisse logida keskkonda Confluence Cloud.	F3
UC-T2	Tavakasutajana soovin näha, kas sisulehest on koostatud allkirjastatud dokument	F4, F6, F8
UC-T3	Tavakasutajana soovin näha, kes on sisulehel dokumenti allkirjastanud	F10
UC-T4	Tavakasutajana soovin otsida kõiki sisulehti, mis on allkirjastatud dokumendina	F39

Dokumendi koostaja põhised kasutajalood on järgnevalt kirjeldatud. Mahukuse tõttu on tabel (Tabel 17) leitav Lisa 7 alt.

Dokumendi kinnitaja põhised kasutajalood on kirjeldatud järgnevas tabelis (Tabel 7):

Tabel 7. Dokumendi kinnitaja põhised kasutajalood.

Kood	**Dokumendi kinnitaja põhine kasutajalugu**	**Nõue**
UC-R1	Dokumenti kinnitajana soovin näha mind on määratud avatud sisulehel dokumenti allkirjastaks	F22
UC-R2	Dokumenti kinnitajana soovin sisulehte allkirjastada	F24, F25, F26, F30
UC-R3	Dokumenti kinnitajana soovin allalaadida allkirjastatud digitaalset konteinerit	F28
UC-R4	Dokumenti kinnitajana soovin otsida kõiki sisulehti, milles on mind määratud allkirjastajaks	F40

Admistraatori põhised kasutajalood on kirjeldatud järgnevas tabelis (Tabel 8):

Tabel 8. Admistraatori põhised kasutajalood.

Kood	**Admistraatori põhine kasutajalugu**	**Nõue**
UC-A1	Admistraatorina soovin installeerida rakendust Confluence Cloud keskkonda	F1, F2, MF1
UC-A2	Admistraatorina soovin välistada kasutajaid, kellel puuduvad õigused kasutada installeeritud rakendust.	F5
UC-A3	Admistraatorina soovin määrada, millises sisuruumis (*Space*) on installeeritud rakendus kasutusel.	F7
UC-A4	Admistraatorina soovin tühistada pooleni oleva dokumenti allkirjastamist	F19
UC-A5	Admistraatorina soovin luua kasutajagruppe volituste haldamiseks	F43
UC-A5	Admistraatorina soovin määrata kasutajale õigust dokumente allkirjastada	F45

3.1.7 Tulevikus loodavad funktsionaalsused

Selleks, et hoida projekti arendust hallatavana, on mõned planeeritavad funktsionaalsused jäetud väljapoole antud lõputöö skoopi. Siia hulka kuuluvad dokumendi komplekti allkirjastamine, töövoogude allkirjastamine, tegevuste ajalugu, kuna need funktsionaalsused ei ole vajalikud baasrakenduse loomisel vajalikud, kuid tulevikus aitavad laiendata rakenduse funktsionaalsust. Samuti kuuluvad siia administraatori jaoks mõeldud funktsionaalsused, kuna need otseselt ei mõjuta tavakasutaja kogemust.

3.2 Tehnoloogia valik

Käesoleva lõputöö skoobis luuakse Restful rakendus, mille eesmärk on pakkuda tagarakendusena (*backend application*) teenuseid dokumentide digitaalseks allkirjastamiseks tarkvaraplatvormil Confluence Cloud. Sellest lähtuvalt on lõputöö skoobis vaatluses tänapäeval levinud arendustehnoloogiaid, mis võimaldavad luua kaasaegset API-põhist tagarakendust. Antud peatükis kirjeldab ja põhjendab lõputöö

autor, milliseid tehnoloogiad on kasutatud lõputöös loodavas API-lahenduse realiseerimisel.

Tehnoloogia valikul on arvestatud, et rakenduse äriloogika asub tagarakendusena API-põhises tagarakenduses ning kasutajaliides koos kasutajavaadetega teostatakse jätkuarendustes eesrakendusena tarkvaraplatvormil Confluence Cloud. Samuti on API-liidese tehnoloogia valimisel arvesse võetud, et tulevikus on kavas liidestada tagarakendus ka teiste platvormide ja tarkvaradega.

Tänapäeval on veebitehnoloogias kõige levinum arhitektuuristiil erinevate süsteemide omavaheliseks liidestamiseks REST[1] [26]. REST API'd moodustavad üle 70% kõigist maailma API'dest ning on raske leida ühtegi suuremat veebirakendust, mis ei kasutaks REST API'd oma lahenduses. REST on lühend terminile *Representational State Transfer* [27], mis pakub alternatiivi SOAP'ile (*Simple Object Access Protocol*) [28]. REST'i eelis on selle kergem kasutatavus, paindlikkus andmeformaatide osas ning on enamusel juhtudel kiirem ja jõudlikum. REST lähenemine võimaldab tulevikus kergema vaevaga arendada lahendusi ka teistele platvormidele ning tagab vajalikku tulevikukindluse.

Lähtuvalt REST arhitektuuristiilist on järgnevalt eraldi analüüsitud tagarakenduse poolseid ja kliendipoolseid eesrakenduse tehnoloogiaid ja nendega kaasnevaid raamistikke ning keeli. Valikute tegemisel arvestatakse järgnevate teguritega, mis tulenevad loodavast API-põhisest tagarakenduses ja tuleviku perspektiivist:

- Kulud ja litsentsid – tehnoloogiate valiku puhul tuleb arvesse võtta, kas tegemist on vabavaraga, taskukohase teenustasuga või suurematele ettevõtetele mõeldud kallima tehnoloogiaga.
- Paindlikkus – loodavas lahenduses tuleks võtta kasutusele ainult tehnoloogiaid, mis on võimelised ühilduma ka teiste populaarsete tehnoloogiatega. Oluline on orienteeritus muutustele ja edasiarenemisele. Muutuvas maailmas on paindlikkus eelduseks nii tarkvara arenduste edukuse kui ka infosüsteemide jätkusuutlikkuse aspektist.

[1] *Representational State Transfer* – veebiteenusega suhtlemise liidese arhitektuur

- Konteineriseerimistehnoloogia[1] – tehnoloogia peab võimaldama ehitada kergeid ja õhulisi rakendusi, mida on optimaalne ja efektiivne käidelda virtualiseeritud konteinerkeskkondades.
- Skaleeritavus – tehnoloogia peaks kokku sobima erinevate orkestreerimissüsteemidega, mis tagavad skaleeritavuse [28]. Eesmärk on tulla toime suurenenud (*upscale*) või vähenenud (*downscale*) töökoormusega, et antud vajadused saaksid võimalikult optimaalselt rahuldatud.
- Infrastruktuur – valitav tehnoloogia peab olema lihtsasti seadistav ja käideldav pilveplatvormidel (*Cloud computing*[2]) ning ei tohiks nõuda spetsiaalset riistvara koos füüsilise serveriga (*On-premise*[3]). Sellega kaasneksid täiendavad IT-kulud, sest riistvara koos tarkvaraga nõuavad pidevat hooldust, tuge ning valmisolekut hädaolukordadega tegelemiseks ja probleemide lahendamiseks.
- Tugi – Tehnoloogia peab pakkuma pikaajalist tuge (LTS, *Long-Term Support*[4]), mille eesmärk on säilitada toodet või rakendust pikema aja jooksul. Pikaajaline tugi sisaldab värskendusi, mis käsitlevad tehnilisi probleeme, vigade parandusi ning turvapaiku. Kui pikaajalist tuge tehnoloogiatele ei pakuta, on neid riskantne rakenduse ehitamisel kasutada.
- Küpsus – tehnoloogia peab olema piisavalt küpsus ning stabiilne, et tagada toodangu keskkonnas töökindluse ning usaldusväärsuse.
- Kogemus – lõputöö tegemiseks on piiratud aeg, sellepärast tuleb tehnoloogiate valimisel lähtuda ka autori oskustest ja kogemusest nendega. Uue tehnoloogia omandamine ning mitte-ennustavate probleemidega tegelemine võib pikendaks lõputöö valmimise aega. Seega on tehnoloogia valimisel oluline silmas pidada selleks õpitavat aega kui ka autori enda kogemust.
- Turvalisus – loodav rakenduse puhul tuleks eeldada tundlikke isikuandmete käitlemist ning seetõttu on oluline arvestada turvalisuse aspektiga. Enamikel

[1] Konteinerid on operatsioonisüsteemi virtualiseerimise meetodi tulemus, mis võimaldavad efektiivsemat füüsiliste ressursside kasutust host seadmel ning madalamat ajakulu uue guest instantsi loomisel.

[2] Pilveandmetöötlus on serveri või rakenduse majutusteenuste pakkumine üle interneti.

[3] Füüsiline server, mida haldab klient ise. See hõlmab operatsioonisüsteemi paigaldamist, tulemüüri seadistamist, tarkvara uuendamist ning serveri tervise monitooringut.

[4] Pikaajalise toega tarkvara versioon. Tavaliselt on perioodi pikkuseks 2 kuni 3 aastat.

raamistikel on oma lähenemine rakenduse turvalisusele ning nende võimalustest peaks teadlik enne otsuse langetamist.

- Dokumenteeritus – tehnoloogia valikul on mõistlikum kalduda tehnoloogiate poole, mis omavad head tuge, dokumentatsiooni ning mille puhul on probleemidele võimalik vastuseid leida.
- Kogukond – valiku langetamisel tuleb veenduda, et tehnoloogial on aktiivne kogukond ning pidev tehnoloogia arendustsükkel.

3.2.1 Tagarakendus programmeerimiskeele valik

Tänapäeval on võimalik valida programmeerimisekeelt paljude valikute seast. Aastal 2021. viis tarkvara arendajatele suunatud keskkond Stack Overflow[1] läbi küsitlus arendajate seas, milles osales üle 80 000 arendaja [30]. Küsitlusest tuli välja, et kõige populaarsemad tagarakenduse poolsed programmeerimiskeeled on Java, C#, PHP, Python ja Node.js. Järgnevalt on toodud välja nende keelte lühikirjeldused:

- PHP – avatud lähtekoodiga serveripoolne skriptikeel, mida sageli kasutatakse veebiarenduseks. Üldkasutatav keel, mida saab kasutada paljude arenduste, sealhulgas graafiliste kasutajaliideste tegemiseks [31]
- Java – objekt-orienteeritud keel, mida interpreteerib JVM ehk *Java Virtual Machine* ning jookseb mitmetel platvormidel. Tüübi- ja klassipõhine objektorienteeritud keel, mis kompileeritakse baitkoodi käsukomplekti ja binaarvormingusse [32].
- C# – objekt-orienteeritud ning tüübikindel keel. Võimaldab ehitada erineva keerukusega rakendusi ning on kõrge koormuse taluvusega ning töötavad vaid platvormidel, mis toetavad .NET keskkonda [33].
- Python – avatud lähtekoodiga tõlgendatud, objektorienteeritud kõrgetasemeline programmeerimiskeel, millel on dünaamiline semantika. Selle kõrgetasemelised sisseehitatud andmestruktuurid koos dünaamilise tippimise ja sidumisega, muudavad selle väga atraktiivseks nii rakenduste kiireks arendamiseks kui ka kasutamiseks skripti- või liimkeelena olemasolevate komponentide ühendamiseks. [34]

[1] https://stackoverflow.com [30]

- Node.js – avatud lähtekoodiga ja platvormist sõltumatu JavaScripti käivituskeskkond. Node.js on ehitatud Google Chrome JavaScripti V8 mootori peale. Node.js JavaScript'i käituskeskkond, mis võimaldab arendajatel luua JavaScripti kasutades serveripoolseid rakendusi [35]. Tänapäeval kasutatakse laialdaselt Node.js rakenduste loomisel TypeScript'i programmeerimiskeelt, mis täiendab JavaScript'i.

Järgnevalt on välja toodud tagarakenduste poolsete keelte võrdlus kasutades võrdluspunktideks lõputöö autori kogemust ning programmeerimiskeele õppimiskeerukust (Tabel 9). Kõik tabelis välja toodud keeled on hästi dokumenteeritud ning laialdaselt kasutuses veebirakendustes.

Tabel 9. Programmeerimiskeelte võrdlus

Keel	**Kogemus**	**Õppimiskeerukus**
PHP	Rahuldav	Madal
Node.	Väga hea	Keskmine
Java	Hea	Keskmine
C#	Hea	Keskmine
Python	Kehv	Madal

Kõik loetletud programmeerimiskeeled on paindlikud, turvalised ja väga hea dokumentatsiooniga. Arvestades seda, et uue keele õppimiseks on lõputöö skoobis piiratud aeg, siis edaspidi on mõistlik kaaluda neid keeli, millega neist lõputöö autoril on kõige rohkem kogemust. Võttes arvesse autori on senist head kogemust keeltega Node.JS, Java ja C#, siis piirdutakse edasiselt ainult nimetatud keeltega, sest teiste keeltega pole piisavalt kogemust ja juurde õppimine võib võtta liialt palju väärtuslikku ajaressurssi.

Lõputöö autori arvamusel on keerukus kõigil kolmel keelel keskmine. Lõputöö autoril on seni kõige rohkem kogumusi Node.js keelega, millega on autor 8 aastat jooksul loonud igapäeva töös mitmeid keerukaid tagarakendusi. Kuigi see ei mõjuta suurel määral tehtud otsust, sest autoril on kõigi kolme valitud keeltega kogemusi.

Kõigil kolmel keelel on mitmeid raamistikke ning tööriistu, mida tuleb võtta arvesse valiku tegemisel. Arvesse tuleb samuti võtta keele valikuga kaasnevad ökosüsteemi, kogukonda ja programmeerijate kättesaadavust [36]. Lõputöö raames vaadeldakse eeskätt REST API raamistikke, mis on mõeldud API-rakendusliideste loomiseks. Node.js raamistikke on saadaval laias valikus ning on neid loodud erinevalt eesmärkidel [37]. Esile tasuks tõsta NestJS, mis on aastate jooksul saanud tänu laiale kasutajaskonnale piisavalt küpseks. NestJS loomisel ja arhitektuuri disainimisel on inspiratsiooniks võetud Java Spring raamistik ja JavaScript'i brauseripoolne raamistik Angular [38]. Nende kolme raamistikku koos kasutamine võimaldab kasutada sarnast arhitektuurilist lähenemist ning taaskasutada sarnasi valmislahendusi. NestJS eeliseks on põhjalik dokumentatsioon koos hästi kirjeldatud näidetega.

Microsofti C# keelel on kaks väga levinud ja populaarset raamistikku - aastal 2002 loodud .NET Framework, mille viimane versioon on 4.8 [39] ning .NET Core, mis võeti kasutusele aastal 2016 ning millel versioon 7 on hetkel viimane [40], [41]. Aastani 2022 oli veel C# keelel kõige laialdasemalt kasutatud .NET Framework raamistik. Stack Overflow 2022 arendajate uuringust tuli välja, et arendajate lemmik raamistikuks C# maailmas on nüüdseks saanud .NET Core [42]. Kuna .NET Core on võrreldes .NET Framework-iga uuem ja üha rohkem populaarsust koguv ning .NET Frameworki edasiarendus, siis valikusse otsustati võtta .NET Core.

Java populaarsemateks raamistikeks on Spring, Struts ja Hibernate [43]. Siiski paistab nende kolme seast enim välja Springi raamistik, mida nimetatakse raamistike raamistikuks. Just nagu Java ja C# on süntaksi ja semantika poolest väga sarnased, on ka nende suurimad raamistikud sarnased. Nii Spring kui ka .NET Core on mõlemad väga hea dokumentatsiooni ja laialdase kasutajaskonnaga. Mõlemad on paindlikud ja turvalisuse aspektis võrdväärsed. Samuti on autoril kogemust mõlema keele ja raamistikuga.

Kõigil kolmel keeles arendatud välja tööriistad, mis eeskätt aitavad pakettide haldusega, projekti ehitusega ning aitavad kerge vaevaga jooksutada teste. Node.js keelel on põhiselt kasutasel NPM ja Yarn tööriistad [44]. Mõlemad tööriistad on küpsise saavutanud, on paindlik ning neid kasutavad iga päev sajad tuhanded JavaScript'i arendajad. Java'l on aja jooksul arenenud põhjalik tugi tööriistade nagu Maven'i ja Gradle'i näol. Raamistikus .NET on analoogne lahendus NuGet pakettide haldur, kuid üldjuhul peetakse küpsemateks tööriistadeks Java variante. Lisaks on paljud Node.js ja Java jaoks loodud abivahendid avatud lähtekoodiga ning tasuta kasutatavad.

Arvestades lõputöö teemat ja skoopi, otsustas autor Node.js ja NestJS raamistiku kasuks. Otsuse tegemisel kaaluti eelnevalt loetletud lõputöös tehnoloogia lähtetingimusi ning NestJS ja TypeScript'iga suuremat kogemust. Määrav tegur oli senine autori kogemus.

3.2.2 Tagarakenduse poolsed raamistikud

Tagarakenduse (*backend*), või ka teenusrakenduse, valikul tuleb võtta arvesse mitmeid faktoreid: riistvara, operatsioonisüsteem, raamistik, programmeerimiskeel ja andmebaas. Andmebaasi valikut kirjeldatakse hilisemas peatükis. Erinevaid komponente ja nende tehnoloogiaid on infotehnoloogia tööstuses ajalooliselt käsitletud grupeeritult ning ühe taolise grupi nimetus on *technology stack* ehk tehnoloogia pinu.

Tänapäeval on veebitehnoloogiad jõudsalt edasi arenenud ning olulise arenguna on viimase kümne aasta jooksul on hakatud jõudsamalt eraldama teenuste- ning kliendipoolset loogikat. Selline lähenime võimaldab iseseisvalt ja lahus arendada tagarakenduse poolset äriloogikat ning kliendipoolset eesrakendust. Seetõttu on vähenenud pinude kasutamine ning aina kasutatakse erinevaid raamistikke.

Autor on töökogemustes puudunud kokku järgmiste populaarsete tehnoloogia pinudega [45]:

- LAMP pinu – antud pinu all mõeldakse eelkõige Linuxi opereerimissüsteemi, Apache HTTP Serverit, MySQL RDBMS'i ning PHP programmeerimiskeelt. Kogu pinu on vabavara-põhine ning on üks kõige esimestest populaarsust kogunud pinudest. See on samuti väga paindlik, kiiresti arendatav ning võimaldab kasutada ka teisi vabavaralisi tehnoloogiaid. Opereerimissüsteem võib olla ka

Windows- või Mac-põhine, andmebaasina võib kasutada PostgreSQL-i ning keele asendada Perl'i või Python'iga.

- MEAN pinu – sarnaselt LAMP pinule vabavara põhiline pinu, mida kasutatakse laialdaselt dünaamiliste veebilehtede ehitamiseks. Traditsioonilise relatsioonilise andmebaasi asemel on kasutusel NoSQL, mille populaarne näide on MongoDB. E, A ja N tähistavad omakorda Express.js-i, Angular JS-i ja Node.js.-i
- Java pinu – Java'l põhinev tehnoloogia pinu, mille puhul on veebiteenuse poolel kasutuses Java põhised raamistikud nagu Spring või Spring Boot, Grails ja muud. Serveri poolel on kasutuses lahendused nagu JBoss või Apache Tomcat Java pinu põhjal pole oluline andmebaasi ega kliendipoolse rakenduse tehnoloogiate valik.

Kaasaegsetes veebirakenduses on eraldatud tagarakenduse äriloogika ning kliendipoolset eesrakendus. Seda on võimaldanud brauserite jõudluse kasv, mis võimaldab rohkem kliendipoolset koodi jooksutada [46]. Selline lähenemine annab veebirakenduse ja arendusprotsessile mitmeid eelised, millest mõned on järgnevalt loetletud [47]:

- Skaleeritavus – koodiosa on jaotatid kaheks iseseisvaks osaks ning see võimaldab koodi paremini optimeerida. Samuti saab eraldi suurendada tagarakendusele ja esirakendusele jagatavad ressursse. See võimaldab paremini ressursse optimeerida ning efektiivsemalt kasutada.
- Lihtsam uuendamine – raamistiku uuendamine võib olla keeruline väljakutse. Taga- ja esirakenduse lahus hoidmine vähendab riski, et peale uuendamist on veebirakenduses tõrkeid. See lihtsustab vea põhjuse leidmist ning lahendamist.
- Kiirem arendusprotsess – taga- ja esirakendust saab iseseisvalt arendada ning testida. Vähenevad olukorras, kus tarkvara arendus peab ootama, millal taga- või esirakendust arendus jõuab järgi.
- Modulaarsus – suurt projekti on lihtsam jagada väiksemateks projektideks, kui taga- ja esirakendus on lahus ning nende vahel on nõrgad sõltuvused.

Eelnevalt välja valitud programmeerimiskeelte Node.js, Java ning C# tuntumad tagarakenduse poolsed raamistikud on [45]:

- Spring – Javal põhinev raamistik, mis on avatud lähtekoodiga ning pakub väga põhjalikku infrastruktuuri tuge veebiteenuse arenduses. Springi paremaks

kasutamiseks loodi Spring Boot projekt, mis lubab kerge vaevaga eelnevalt konfigureeritud Springi raamistikuga rakendust luua.

- Micronaut – Java põhine mikroraamistik, mis pakub rohkeid võimalusi võrreldes mitmete teiste raamistikkega [49]. Hea valik juhul, kui soovitakse rohkem funktsionaalsusi, kuid ei soovita võtta midagi nii põhjalikku nagu Spring Boot.
- .NET – .NET pinu hulka kuuluv suletud lähtekoodiga raamistik, mis omab põhjalikku dokumentatsiooni, head infrastruktuuri tuge ning võimaldab erinevate veebirakenduste kiiret arendamist.
- NestJS – üks kiiremini kasvavaid Node.js'i raamistikke tõhusate ja ettevõtte tasemel tagarakenduste loomiseks [50]. Nest.js on loodud kaasaegse Typescript'i baasil, mis võimaldab arendajatel luua skaleeritavaid, kergesti hooldatavaid ja testitavaid rakendusi. NestJS kasutab sisemiselt HTTP päringute töötlemiseks Express.js raamistikku.
- Express.js – nõuab väga vähe vaeva ning kujutab endast minimaalset paindlikku raamistikku, mis kasutab efektiivselt oma ressursse. Sobib keskmiselt keerukamate veebirakenduste loomiseks. Raamistik järgib MVC[1] disainimustrit.

Tabel 10. Tagarakenduse poolsete raamistike võrdlus

	Paindlikkus	**Kogemus**	**Turvalisus**
Spring	Väga paindlik	Hea kogemus	Omab piisavalt lahendusi
.NET	Väga paindlik	Keskmiselt kogemust	Omab piisavalt lahendusi
NestJS	Väga paindlik	Rohkelt kogemust	Omab piisavalt lahendusi [46]
Express.js	Keskmiselt paindlik	Rohkelt kogemust	Omab keskmisel lahendusi [47]

Võttes aluseks eelnevat tabelit ning eelnevas peatükis tehtud järeldust programmeerimiskeele osas, on mõistlik raamistiku valik NestJS. Node.js on üks lõputöö autori eelistatud programmeerimiskeeltest ning seda toetav NestJS on end tõestanud kui populaarne ja eelistatud raamistik.

[1] Model View Controller ehk mudel-vaade-kontroller

Raamistik NestJS pakub tuge API päringute tegemisel kasutamaks teisi Node.js-põhiseid HTTP[1] raamistikke. See annab võimaluse kasutada teiste raamistikke tugevusi ning funktsionaalsusi, mis on seotud HTTP päringute töötlemiseks.

Järgnevalt on omavahel võrreldud mõningaid Node.js-põhised HTTP raamistikke, millega on võimalik laiendata NestJS raamistikku:

- Express.js – paindlik ning robustne veebirakenduse raamistik, mis on loodud REST API mugavaks loomiseks. Raamistik pakub rohkelt funktsioone HTTP päringute töötlemiseks, suunamisteks ning optimeerimiseks [53].
- Fastify – on üks kiiremaid Node.js raamistikke, mis pakub HTTP päringute töötlemisel kõrget efektiivust. Raamistik keskendub eelkõige jõudlusele ning minimaalsusele ning seetõttu on raamistikul võrreldes Express.js'ga vähem funktsionaalsusi [54].

Käesoleva lõputöö arendamisel on võetud kasutusele NestJS, sest on laialdaselt levinud raamistik, mis võimaldab kiiret projekti ehitamist ja konfigureerimist mitmete saadaval olevate pakettidega.

3.2.3 Kliendipoolne eesrakenduse tehnoloogia

Järgmises arendusetappis loodav kliendipoolne eesrakendus luuakse tarkvaraplatvormile Confluence Cloud ning ehitatakse *plugin*'ina. Sellest lähtuvalt tuleb eesrakenduse tehnoloogia valimisel arvestada tarkvaraplatvormi piirangutega ning võimaldavate tehnoloogitega.

Atlassian'i pilveplatvormidel on alates aastast 2019 kasutusel tehnoloogia *Atlassian Forge*, ehk Forge [55], [56]. Tehnoloogia programmeerimisekeel on JavaScript ning põhineb populaarsel JavaScript'i React'i paketil [57]. Forge võimaldab ehitada tarkvaraplatvormile Confluence Cloud funktsionaalsust laiendatavaid *plugin*'eid ning pakub lisaks sisseehitatud *SaaS* eesrakenduse automaatset paigaldamist, mitut arenduskeskkonda ning Atlassian API tuge koos automaatse autentimise lahendusega.

[1] Andmevahetusprotokoll - kasutatakse internetis dokumentide vahetamiseks

Forge pakub rakenduste kasutajaliidese loomiseks ainult kahte võimalust: kohandatud kasutajaliides, ehk *Custom UI* ja kasutajaliidese komplekt, ehk *UI kit*. Need valikud erinevad interaktsioonide ja visuaalsete funktsioonide keerukuse poolest. Järgnevalt on kirjeldatud mõlemat varianti [58]:

- *Custom UI,* ehk kohandatud kasutajaliides – võimaldab Atlassiani toodetes kuvada rakenduse kohandatud kasutajaliidest kasutades selleks React paketiga loodud vaateid. Mõeldud eelkõige keerukamate rakenduste loomiseks ning arendajatele suurema paindlikkuse tarvis. Rakendus töötab kliendipoolel ning liidestamine tagarakendusega on kas API-rakendusliideste kaudu või integreerides Forge'i tagarakendusega, kutsudes välja tagarakenduse funktsioone *Forge Bridge* kaudu [59]. *Custom UI* abil saab luua kohandatud vaateid, kasutades staatilisi ressursse, nagu HTML, CSS, JavaScript, ikoonid ja pildid.
- *UI kit*, ehk kasutajaliidese komplekt – mõeldud Forge'i rakenduste jaoks, millel on lihtsamad kasutusjuhud. Loodavad vaated käivitakse ja genereeritakse tagarakenduses ning seejärel saadetakse tagasi brauseripoolele. *UI kit* töötab *SSR* [1] põhimõttel. Sobilik näiteks rakenduse konfiguratsioonihalduse loomiseks.

Tulenevalt tarkvaraplatvormi Confluence Cloud'i tehnoloogia piirangutest on vajalik kasutada eesrakenduse loomisel Atlassian Forge'i tehnoloogiat koos React'i paketiga. Autor omab kogemust nii React'i kui ka Forge'i tehnoloogiatega ning React on samuti eelistatud eesrakenduse valikuna.

3.2.4 Transpileri valik

Tagarakendus ja eesrakendus kasutavad mõlemad JavaScript'i programmeerimiskeelt, mis ei toeta staatilist tüüpimist[2] ning tehtud vead võivad alles ilmneda rakenduse *runtime* keskkonnas. Selle puuduse põhjusel loodi Microsoft'i poolt aastal 2002 TypeScript, mis on avatud lähtekoodiga JavaScript'il põhinev programmeerimiskeel [60]. TypeScript'i tehnoloogia võimaldab JavaScript'ile lisada staatiliste tüüpide definitsioonid ning seeläbi aitab vältida vigade tekkimist ja kiirendab andmete kasutamist arenduse käigus.

[1] SSR ehk *Server-Side Rendering* – veebisaiti sisu luuakse serveris ja saadetakse seejärel brauserisse.

[2] staatiline tüüpimine, mis võimaldab tüübivigade avastamist enne programmi käivitamist.

TypeScript ei ole loetav veebibrauserite ega Node.js keskkonna poolt, vaid enne tuleb transpileri[1] poolt teisendada TypeScript'i koodi JavaScript'iks.

TypeScript võimaldab määrata tüüpe, liideseid, loendeid, funktsioonide parameetreid ja nende tagastusväärtuse tüüpe ning märkida millised väljad või parameetrid on mittekohustuslikud [61]. TypeScript'i toetav IDE võib sisestamise ajal pakkuda võimalikke valikuid, mida antud objekt, funktsioon või komponent võib sisaldada. See kiirendab arendamise protsessi ja aitab arendajatel üksteise koodist hõlpsasti aru saada. TypeScript'i programmeerimiskeeles arendamine võib võtta rohkem aega, kuid tulevikus on lihtsam rakendust edasi arendada ja tagada koodi kvaliteeti.

TypeScript on JavaScript'i arendamisel väga populaarne ja tugevalt soovitatud ning seetõttu tehti valik kasutada nii tagarakenduse ja eesrakenduse arendamisel TypeScripti [62].

3.3 Andmebaasi valik

Andmebaasid jagunevad mitmetesse kategooriatesse [63]: puustruktuuriga, relatsioonilised, objekt-orienteeritud, dokument-orienteeritud ning võti-väärtus andmebaasid. Tagarakendustes kasutatavatest andmebaasidest tehakse valik eelkõige relatsiooniliste, dokument-orienteeritud ning võti-väärtus andmebaaside vahel. Lõputöö raames on vaatluse alla võetud avatud lähtekoodiga andmebaasid ning analüüsitud nende erinevusi.

Relatsiooniliste andmebaaside puhul on andmete puhul tegemist spetsiifiliste väljadega tabelites, mis on omavahel seotud defineeritud suhetega. Kasutatakse eelnevalt struktureeritud andmete puhul. Relatsiooniliste andmebaaside suureks plussiks on andmetüüpide rangus, mis vähendab vigade esinemist. Relatsiooniliste andmebaasi iseloomustab kõrge usaldusväärsuse ja terviklikkuse, mille tagab ACID printsiipi järgimine. ACID tuleneb terminitest [64]:

[1] Transpiler - programm, mis teisendab ühe kõrgema taseme keele sisendkood teise kõrgema taseme keele väljundkoodiks

- atomaarsus (*Atomicity)* – ülekanne peab olema jagamatu, saabudes kehtivalt (juriidiliselt) lõpetatuna ja vastasel korral tühistatuna, viimane taastab andmete esialgse seisu, mis loogikaterminites tähendab kõik või mitte midagi.
- kooskõlalisus (*Consistency*) – sobivalt valitud andmete uuendamisreeglid peavad välistama vigade ja ebakõlade teket eristatavate ja kooskõlas olevate andmete haldamisel.
- eraldatus (*Isolation*) – üheaegset juurdepääsu (*concurrency*) andmetele erinevate protsesside poolt peab reguleerima, et vältida konflikte ja DBMS[1] väärastumise/rippuma jäämise (*deadlock*) situatsioonide vältimiseks.
- kestvus (*Durability*) – andmete juhusikke kaotusi peab vältima või viima minimaalseks, lisades viimase kehtiva seisundi, milline süsteemil oli enne talitlushäiret, ehk taastamisvõimaluse.

Järgnevalt on välja toodud mõned tuntumad relatsioonilised andmebaasid:

- Sqlite – andmebaasi suurus limiteeritud, kuid kerge õppida. Nii kirjutamine kui ka lugemine toimub kiirelt ning seda saab kasutada mitmete erinevate arenduskeskkondade ja keeltega [65]. Ei võimalda kasutajate haldust läbi ligipääsu õiguste [66]. Võrreldes teiste relatsiooniliste andmebaasidega on vähe andmetüüpi ning seetõttu ta ei sobi programmeerimiskeeltega, millel ei ole staatilisele tüübikontrolli.
- MySQL – sarnaselt Sqlite'ile on kiire ning kerge õppida. Lisaks omab MySQL turvalisuse aspekte ning on skaleeritav. Kahjuks vajab litsentsi, kui seda kasutab mitte-vabavaraline kommertslikel põhjustel loodud aplikatsioon ning on piiratud funktsionaalsustega [67].
- PostgreSQL – avatud lähtekoodiga ning suure kasutajate baasiga, mille tõttu omab ka väga suurt tuge. PostgreSQL'l on head omadused andmete terviklikkuse kaitsmisel tehingu tasemel ning see vähendab haavatavust andmete kahjustamise suhtes [68]. MySQL'ga võrreldes sisaldab PostgreSQL rohkem funktsioone ja laiendusi ning seetõttu võib kannatab päringute kiirus. Samuti ei sooritada lihtsaid operatsioone väga efektiivselt ning jääb selle poolest alla MySQL'ile [66].

[1] DBMS (*Database Management System*) ehk andmebaasi haldussüsteem

Dokument-orienteeritud andmebaaside puhul on tegemist andmetega, mida salvestatakse JSON[1], BSON[2] või XML[3] kujul. Kasutatakse struktureerimata andmete puhul, kui andmebaasi välimus ei ole ette teada. Järgnevalt on mainitud paar tuntumat dokument-orienteeritud andmebaasi:

- MongoDB – kergelt skaleeritav, väga paindlik, võimaldab paremat protsessimistõhusust ning paindlikke päringuid, kui neid ei tehta üle mitme dokumendi. Viimasest omadusest ilmneb ka MongoDB puudus – andmebaas ei sobi relatsiooniliste andmete jaoks ning ei oma sisemist transaktsioonide haldust. Struktuurse ülesehituse tõttu on andmebaasil kõrge mälu kasutus [69].
- Couchbase – väga sarnane MondoDB'le, kuid omab kergemaid konfigureerimisvõimalusi ning ei vaja välist puhverdamist tänu paremale jõudlusele. Kergemini hallatav läbi kasutajaliides, käsurea liidese või REST API. Seevastu toetab vähem programmeerimiskeeli kui MongoDB [70].

Võti-väärtus andmebaasid on kõige paindlikumad NoSQL andmebaasid, kus salvestatavad andmed pole mitte vaid dokumentide kujul, vaid võivad olla ka numbrid, sõnad, pildid, videod ja palju muud [71]. Andmetele ligipääsemiseks on vaja teada võtmeid, mis neid väärtusi hoiavad. Päringute tegemine on piiratud võimalustega, mis ei võimalda sooritada keerukaid päringuid ega mõningaid andmete agregeerimise funktsionaalsusi. Samuti ei toeta võti-väärtus andmebaasid ACID kontrolle ning andmete usaldusväärsus ja terviklikkus ei ole garanteeritud.

Redis on üks tuntumaid võti-väärtus andmebaase on, mille tugevuseks on selle kiirus nagu teistelgi NoSQL andmebaasidel [72]. Redis võimaldab mitmeid lisafunktsionaalsusi ning on väga kõrge käideldavusega ehk käsitleb mõningaid rikkeid ilma inimeste sekkumiseta. Redis funktsioneerimiseks RAM'i ning on seega tundlik suurte andmete osas. Samuti on problemaatiline turvalisus, kuna Redis ei toeta krüpteerimist ega ligipääsu kontrolli.

[1] Javascript Object Notation - Javascript'il põhinev andmevahetusvorming

[2] Binary JSON ehk binaarne JSON - andmevahetusvorming

[3] Extensible Markup Language – dokumentide märgendamiskeel, mis on nii masin- kui ka inimloetav

Lõputöö andmebaasi valikuks sai valitud relatsioonile andmebaas PostgreSQL. Loodavas andmebaasis on kasutatavad andmed teada ning andmemudelit on võimalik struktuurida ning normaliseerida [73]. PostgreSQL võimaldab samuti kasutada JSONB andmetüüpi, mis võimaldab salvestada ning töödelda struktureerimata andmeid [74]. Lõputöö autoril on rohkem kui üle 15. aasta kogemust PostgreSQL'i andmebaasiga ning 2. aastat kogemust MongoDB'i andmebaasiga ning senisest kogemusest lähtuvalt eelistab autor kasutada PostgreSQL'i andmebaasi.

3.4 Arenduskeskkonna valik

Lõputöö kontekstis loodavate rakenduste koodibaasid peavad lähtuma pikaajalisest perspektiivist ning võimaldama töötada mitmel arendajal samaaegselt sama koodibaasiga. Seetõttu on oluline, et valitav koodihalduse tehnoloogia baseeruks jagatud süsteemil, ehk *DVCS*[1] süsteemil [75]. Tänapäeval on kõige populaarsem jagatud versioonikontrolli süsteemi tehnoloogia Git vabavaraline tarkvara. Paljud versioonihalduskeskkonnad on loonud mitmeid oma pakette ja teenuseid laiendamaks Git'i funktsionaalsust. Kõige laialdasemalt levinud versioonihalduskeskkonnad, mis pakuvad ka tasuta paketi, on Github, Bitbucket ning Gitlab, kes keskendub peamiselt DevOps'i tarkvaraarenduse kultuurile omastele teenustele ja toodetele [76], [82].

Käesolevas lõputöös võeti kasutusele Bitbucket'i versioonihalduskeskkond. Toote omanik on Atlassian, kelle toodete hulka kuuluvad Confluence Cloud ja Jira – üks parimaid arendustöö organiseerimiseks mõeldud tööriistasid [81]. Sel põhjusel on Bitbucket'il väga hea integratsioon Jira'ga ja Confluence Cloud'ga.

3.5 Tagarakenduse haldus

Loodav tagarakendus peab oma teenuste pakkumiseks olema üle interneti kättesaadav ning seetõttu tuleb paigaldada avalikult kättesaadavasse võrguserverisse. Serveri lahenduste valimisel tuleb valida kahe variandi vahel. Esimene variant on traditsioonilisem ning nõuab füüsilise serveri olemasolu, mida kas omatakse ise või renditakse pakkuja käest. Füüsilise serveri puhul räägitakse füüsilisest riistvarast, millele

[1] DVCS - *Distributed Version Control System* ehk jagatud versioonikontrolli süsteem

on antud piiratud hulk mälu, kõvakettaruumi ning ülekandekiirust. Rentimise puhul haldab turvalisust, administreerimist ja muid aspekte serverit rentiv firma [88].

Teise variandi puhul käsitletakse pilveteenuseid ehk servereid, mis ei asu kindla füüsilise riistvaral, vaid eksisteerivad jagatud virtuaalses keskkonnas, millel puudub kindel geograafiline asukoht. Pilveteenused on kogunud populaarsust, sest jagatud ruumi tõttu säästetakse nende pealt raha ning olenevalt veebirakenduse suurusest ja külastatavusest saab maksta rohkem või vähem [88]. Füüsilise serveri puhul kaasnevad kulud ka siis, kui töökoormust ei ole. Lisaks on pilveteenustesse veebirakenduse paigaldus kiirem.

Pilveteenused jagunevad kolme kategooriasse: *SaaS*, *PaaS* ning *IaaS* [89]. Kui füüsilise serveri puhul on võimalik täielikult kontrollida kõiki serveri aspekte, siis pilveteenuste puhul on igal kategoorial teatud osa kontrollist võetud teenusepakkuja poolt üle.

IaaS ehk *Infrastructure as a Service*[1] pakub tellijale kontrolli infrastruktuuriliste elementide üle nagu serverid, arvutivõrk ning andmete hoiustamine. Infrastruktuur asub teenuspakkuja pilveteenus ning füüsilise heaolu eest vastutab teenusepakkuja. Teenuse tellijal on täielik kontroll läbi kasutajaliidese või API kaudu. Tuntumate näidete hulka kuuluvad Amazon Web Service ja DigitalOcean.

PaaS ehk *Platform as a Service*[2] pilveplatvormi teenus, pakub tellijale rakenduste ja tarkvara loomiseks vajalikku raamistikku, tarkvara ja tööriistu ning mitmeid abivahendeid rakenduse administreerimiseks. Teenuse pakkuja vastutab operatsioonisüsteemide, vahevara ning rakenduse keskkonnas jooksutamise eest. *PaaS* on hea variant siis, kui veebirakenduse arendusel on soov keskenduda vaid äriloogika kirjutamisele ning hoida rakenduse üles seadmist võimalikult lihtsana. Tuntumate näidete hulka kuuluvad Heroku, Google App Engine ja Red Hat Openshift.

SaaS ehk *Software as a Service*[3] kujutab endast pilvepõhist teenust, mis omab kindlaid funktsionaalsusi, kasutatakse vaid veebi kaudu ning mille kasutajad ei vastuta ei

---

[1] Pilvepõhine teenus, mis haldab infrastruktuurilisi elemente nagu server

[2] Pilvepõhine teenus, mis haldab infrastruktuurilisi elemente, operatsioonisüsteeme, vahevara ja muud, mis ei ole seotud rakenduse ning äriloogikaga

[3] Pilvepõhine teenus, mille puhul on nii platvorm kui ka funktsionaalsused hallatud teenuse poolt

riistvaraliste ega tarkvaraliste muudatuste eest. Lõputöö konteksti kuuluv Atlassian'i pilveteenused on heaks näiteks *SaaS* teenusest, näiteks nende toode Confluence Cloud. Samuti on Dropbox teenus hea näide *SaaS* teenustest.

Käesolev lõputöös keskendub autor pikaajalisele projektile, mis eelduste kohaselt võib leida laialdast kasutust ning seetõttu võib olla jõudluse tagamine pidev väljakutse. Sellest lähtuvalt peab olema tagatud loodava tagarakenduse skaleeritavus. Lõputöös on autor valinud mikroteenuste-põhise arhitektuuri, mida autor kirjeldab järgmises peatükis. Neid faktoreid ja lähtetingimusi arvestades on sobiv valik *PaaS*, mis võimaldab mugavalt juhtida tagarakenduse teenuseid, mis põhineb mikroteenuste-põhisel arhitektuuril ning lubab arendajal keskenduda vaid oma rakenduse arendamisele ja selle jälgimisele

3.6 Tagarakenduse arhitektuur

Lõputöö skoobis teostatakse ainult tagarakenduse API-rakendusliidese arhitektuur. Eesrakenduse arhitektuuri koos kõigi protsessidega on kindlaks määratud ning teostatud Atlassian'i poolt. Käesolevas alampeatükis selgitatakse, milliseid valikuid tehti tagarakenduse arhitektuuri osas.

3.6.1 Hajutatud arhitektuur

Tänapäeval on tagarakenduste arhitektuuris levinud kaks peamist lähenemist: monoliitne ja mikroteenuste-põhine. Kuigi Monoliitne arhitektuur on pikalt olnud tarkvara arendamises peamine traditsiooniline lähenemine, siis kaasaegsem mikroteenuste arhitektuur hakkas leidma laiemat kasutust aastast 2009, millal ettevõtte Netflix'i[1] alustas mikroteenuste arhitektuuri laialdast juurutamist [90].

Monoliitses arhitektuuris töötatakse rakendus välja ühtse pakendina. [91] Monoliitse teenuse võib enamikel juhtudel jagada mõttelisteks osadeks ent omaette ükski neist osadest määravat praktilist väärtust ei oma. Kuigi monoliitsel arhitektuuril on enamasti loogiliselt kihiline arhitektuur, pakitakse lõplik rakendus kokku ühtsesse monoliitsesse paketti ja rakendatakse ühtsena.

[1] Netflix on Ameerika Ühendriikide meelelahutusettevõte, mis osutab voogedastusteenust.

Mikroteenuste arhitektuur järgib modulaarset lähenemisviisi koosnedes väikestest, sõltumatutest ja autonoomsete teenuste komplektist, mis pakub erinevaid teenuseid. [92] Üks mikroteenus täidab üht spetsiifilist rolli ning on ka praktikas eraldiseisva üksusena käsitletav. Mikroteenuste arhitektuuris on rühm väikeseid sõltumatuid üksusi, mis ühiselt töötab ühe rakendusena. Modulaarne ülesehitus võimaldab uuendada teenuse üksikuid komponente ilma ülejäänud teenuse tööd mõjutamata. Mikroteenuste arhitektuur tuleneb SOA[1] teenusorienteeritud arhitektuurist, kuigi mikroteenuste teenused keskenduvad detailsemalt ärifunktsionaalsuste realiseerimisele. [93]

Järgnevalt on võrreldud peamisi erinevusi traditsioonilise monoliitse ja moodsama mikroteenuste arhitektuuri vahel [94]:

- *Skaleeritavus ja paindlikkus* – iga mikroteenuse rakendus on üles ehitatud iseseisvate teenustena, mis aitab muuta kogu rakenduse skaleerimist lihtsamaks. Seevastu monoliitses arhitektuuris on skaleeritavuse arendajatele väljakutse, kuna tegemist on ühe paketiüksusega ning moodulite kaupa ei ole võimalik skaleeritavust juhtida. Koormuse kasvades on monoliitse teenuse skaleerimine võrdlemisi kulukas, sest puudub võimalus skaleerida üksikuid suurima koormuse all olevaid komponente. Lahendusena tuleks kogu rakendus dubleerida mõnesse teise serverisse ning jagada koormust eri serverite vahel koormusjaoturi abil. [95]
- *Liidestamine ja lihtsus* – monoliitse arhitektuuri esmane arendus on sirgjoonelisem ning lihtsamini hoomatav. Tihe integratsioon eri osade vahel tingib olukorra, kus ei ole tarvis ehitada igale komponendile standardset liidest teiste komponentidega suhtlemiseks. Mikroteenuste omavaheline suhtlus käib üle interneti võrgu, mis lisab keerukust ning toob endaga paratamatult kaasa suurema võrgukoormuse.
- *Tehnoloogiale pühendumine* – mikroteenuste puhul saab kasutusele võtta parema ja uue tehnoloogia uute teenuste ehitamisel või olemasolevate teenuste uuendamisel. Monoliitses arhitektuuris on arendajad sunnitud kasutama ainult ühte tehnoloogiat, olenemata selle piirangutest.
- *Rikete isoleerimine* - tõrgete korral on häiritud ainult konkreetse teenuse töö milles antud tõrge ilmnes ning see ei pruugi mõjutada ülejäänud teenuste tööd. Vastupidi

[1] SOA – *Service Oriented Architecture* – teenustele orienteeritud tarkvara arhitektuur.

monoliitses arhitektuuris võib mis tahes komponendi väärkäitumine tõsiselt mõjutada kogu rakenduse tööd.

- *Koodi haldamine* – mikroteenuste koodialused on jagatud mitmeks koodi üksuseks, mis teeb selle haldamise ja värskendamise lihtsaks. Monoliitsete koodialuste suure suuruse tõttu on arendusmeeskondadel väga raske seda õigesti mõista ja juhtida.
- *Arengud ja juurutamine on pidev* – ühe mikroteenuse mis tahes värskendused ei mõjuta muid teenuseid. See toob kaasa pideva arendamise lihtsuse ning keerukate ja tohutute rakenduste juurutamise. Monoliitsel rakendusel on koodialused suured ja komponendid on üksteisest sõltuvad, mis põhjustab pideva juurutamise ja arendamise piiramist, kuna mis tahes komponendi värskendamiseks tuleb täielik rakendus ümber paigutada.

3.6.2 Andmevahetuse arhitektuur

Veebitehnoloogia maailmas on mitmeid tehnoloogiad API'de omavaheliseks andmevahetuseks. Pikaaegselt on olnud andmete vahetamisese standardiks Microsofti poolt algselt arendatud SOAP[1] protokoll [96]. SOAP on kokkulepitud standard programmide käivitamiseks XML'i abil üle HTTP, mille kaudu saavad teenused vahetada struktuurseid andmeid. Maailmas on paljusi pärandvara süsteemi *(legacy systems*), mis on liidestatud teiste süsteemiga SOAP'i kaudu. [97]. Samuti on tehnoloogia väga levinud finantsteenuste, maksusüsteemide, identiteedihalduse ja telekommunikatsiooniteenuste API-liidestes, mis oma keerukuse ja suuruse tõttu on aeglase uuendamise kõveraga. Nendel põhjustel võib eeldata, et SOAP tehnoloogiat kasutatakse veel pikalt.

Alternatiivne lahendus SOAP'ile on REST [98]. REST on kiiresti saavutanud populaarsust alates 2000. aastast kui Roy Fieldling defineeris ja võttis kasutusele REST uue terminina. REST API arhitektuuristiili on tänapäeval veebitehnoloogia maailmas kõige populaarne suhtlusviis erinevate teenuste liidestamiseks [99].

REST on erinevalt SOAP'st arhitektuursete piirangute kogum, mitte protokoll või standard. REST arhitektuuristiilis tehakse erinevat tüüpi päringuid REST-arhitektuuriga üles seatud erinevatele URI'dele, mis seejärel vastavad sobiva vastusega. Vastuse

[1] *Simple Object Access Protocol* - lihtne objektipöödusprotokoll.

vorming ei ole kindlaks määratud ja oleneb rakenduse tüübist. Levinumad vormingud on JSON, HTML ja XML. Rakenduselt saadud vastus sisaldab tagastuskoodi, mis väljendab vastuse tulemust ning olenevalt päringust küsitavaid resursse. HTTP[1] protokolli kasutamisel on peamiselt tehtavad toimingud GET-, POST-, PUT-, DELETE- ja PATCH-toimingud, tuntud ka kui CRUD-toimingud[2]. Enamjaolt põhineb REST API toimingute loogika andmebaasist CRUD päringute realiseerimiseks.

REST'i arhitektuuristiili järgides ehitatud API rakendusi nimetatakse RESTful API'ks. REST määratleb kuus arhitektuurset piirangut, mille järgmisel saab rakendust nimetada RESTful API'ks [100]. Järgnevalt on need piirangud loetletud:

- Ühtne liides (*Uniform interface*) – erinevate klientide päringud peaksid välja nägema ühesugused, näiteks ei tohi ühel ressursil olla rohkem kui üks URI.
- Kliendi-server (*Client-Server*) – klient ja server peaksid tegutsema sõltumatult ning peaksid üksteisega suhtlema ainult päringute ja vastuste kaudu. Klient ei pea teadma teenuspoolset äriloogikat ja teenuspoolne tagarakendus ei pea teadma esirakenduse kasutajaliidest.
- Olekuteta (*Stateless*) – serveripoolseid seansse ei tohiks olla. Iga päring peaks sisaldama kogu teavet, mida server vajab toimingu sooritamiseks.
- Vahemällu salvestatavad ressursid (*Cacheable*) – serveri vastused peaksid sisaldama teavet selle kohta, kas nende saadetud andmed on vahemällu salvestatavad või mitte. Vahemällu salvestatavad ressursid peaksid saabuma koos versiooninumbriga, et klient saaks vältida samade andmete küsimist rohkem kui üks kord.
- Kihiline süsteem (*Layered system*) – tagarakenduse arhitektuur on enamasti kihiline ning kliendi ja serveri vahel võib olla mitu serverikihti. See ei tohiks mõjutada päringut ega vastust.
- Nõudmisel kood (*Code on demand*) – Vajadusel võib vastus sisaldada käivitatavat koodi, mida klient saab käivitada. On valikuline ning ei kuulu põhipiirangute alla.

[1] *Hyper Text Transfer Protocol* - protokoll teabe edastamiseks arvutivõrkudes

[2] *Create, Read, Update, and Delete* – on akronüüm, mis viitab andmebaasiga süsteemides püsiva salvestamise neljale põhitoimingule

Lõputöö autor valis realiseeritavaks API-rakenduse arhitektuuriks mikroteenuste-põhise arhitektuuri, seda nii eelnevalt loetletud praktilistel põhjustel kui ka lähtuvalt tuleviku perspektiivist. REST API oma tulevikukindluses ja populaarsuses tõttu osutus valituks API-liideste loomiseks. REST printsiipide järgimine sobib hästi mikroteenuste arhitektuuris kasutamiseks, sest serveriga suhtlemiseks ühtsete ressursi identifikaatorite kasutamine võimaldab rakendust mugavalt liidestada ka teiste rakendustega [101]. Mikroteenuste arhitektuuriga sobib hästi REST printsiibi põhimõte, et server ei hoia olekuid ning sessioone, sest nii on võimalik luua teenusest mitu instantsi ning kõik instantsid on võimelised klienti teenindama.

Autori isiklik eelistus on samuti mikroteenuste arhitektuuri, millega on tal heal tasemel kogemus. Samuti on autoril mitmeaastane kogemus erinevate Restful rakenduste loomisel ning seetõttu mõjutas valikut ka isiklik eelistus.

3.7 Üldine arhitektuur

Lõputöö kontekstis loodava lahenduse üldine arhitektuur koosneb järgnevatest komponentidest:

- Esirakendus – Atlassian Forge
- Tagarakendus – REST API NestJS raamistikul
- Andmebaas – PostgreSQL
- Versioonihaldus – Git
- *Paas* – mikroteenuste-põhine arhitektuur
- Välised teegid – React ning Node.js teekid

Loodava rakenduse juures realiseeritakse iseseisvalt eesrakenduse ja tagarakenduse arendus. Nii tagarakendusel kui ka eesrakendusel on oma versioonihaldus, välised teegid, automaattestimine ja lisaks kohalikule arendusele *staging environment* ehk testimiskeskkond ning *production environment* ehk reaalne keskkond. Tagarakenduses on igal mikroteenusel oma lokaalne andmebaas ning samuti igas keskkonnas on andmebaasid eraldatud lokaalsed. Andmebaaside eraldamine keskkonna põhiselt on vajalik, et hoida lahus testimiskeskkonnas loodavaid testandmeid reaalsetest andmetest, mis luuakse lõppkasutajate poolt.

Lõppkasutaja suhtleb tagarakenduse teenusega kasutades tarkvavaplatvormil Confluence Cloud installeeritud eesrakendust, mida serveeritakse Atlassian'i pilveserverites, mis põhineb *SaaS* arhitektuuril. Eesrakendus on integreeritud eraldiseisva tagarakandusega ning suhtluseks kasutakse HTTP veebiprotokolli, mis järgib REST API põhimõtteid. Tagarakenduse hoolitseb andmete ning äriloogika rakendamise eest.

3.8 Tulevikus loodav kasutajaliides

Antud lõputöö kontekstis loodaval tootel on kasutajakogemus, ehk *UX* (*user experience*), kõige tähtsam faktor. Kasutajakogemuses koos toote kasutatavusega (*usability*) on peamised faktorid, millest aga sõltub toote edukus turul ja ning lõppkasutajate arv [102]. Toote omadusi disainides on seetõttu oluline järgida järgnevaid tegureid:

- Kasulik – toode pakub kasutaja probleemidele lahendust ning täidab kasutaja eesmärki.
- Kasutatav – toode võimaldab kasutajal võimalikult efektiivselt, kiirelt ja mugavalt saavutada tema eesmärki.
- Leitav – nii toode ise kui ka selles sisalduv info on kergesti leitav.
- Väärtuslik – toode peab pakkuma väärtust nii ettevõttele kui ka kasutajale.
- Usaldusväärne – toode näeb usaldusväärne välja, täidab lubatud eesmärki, on töökindel ning pakutav info on täpne.

Kõikidele nendele teguritele tuleks tähelepanu pöörata, et lõputöö kontekstis loodav toode oleks turul edukas. Järgnevalt on kirjeldatud toote tegureid, mida tuleks arvestada toote disainimisel.

3.8.1 Kasutajaliidese (UI) tehnoloogia

Lõputöö skoobi jätkuarendustes on planeeritud luua kasutajaliides, mis ehitatakse tarkvaraplatvormile Confluence Cloud ning kasutades tehnoloogiat Forge ning React paketi. Alamlõigus „3.2.3 Kliendipoolne eesrakenduse tehnoloogia" on kirjeldatud kahte võimalust, mida Forge pakub rakenduse kasutajaliidese loomiseks: kohandatud kasutajaliides, ehk *Custom UI* ning kasutajaliidese komplekt, ehk *UI kit.* Antud lõputöö kontekstis on valitud kasutajaliidese lahenduse realiseerimiseks Forge'i kasutajaliidese komplekt, ehk *UI kit.* Forge *UI kit* komplektis on saadaval laias valikus kasutajaliidese komponente, mida on võimalik teatud määral kohandada [103].

3.8.2 Kasutajaliidese (UI) disain

Kasutajaliides, ehk *UI,* keskendub sellele, kuidas kasutaja veebilehte kasutab ning kuidas kasutaja veebilehte näeb. Kasutajaliidese osaks on ka konkreetsed interaktiivsed elemendid, mille abil saavad kasutajad nii-öelda suhelda veebilehega (nupud, menüüd, vormid, lingid). Kasutajaliidese ülesanne on täita kasutajakogemuse eesmärke ning seega kasutajaliidest võib vaadata kui vahendit, mis aitab saavutada parimat kasutajakogemust.

Kasutajaliidese disainimisel on oluline arvestada kasutajakogemus (*UX*), sest sellest oleneb, kuidas lõppkasutaja tunnetab toodet. Jakob Nielsen on loetletud 10 printsiibi, mille järgimine aitab saavutada paremat lõppkasutaja kasutajakogemust [104]:

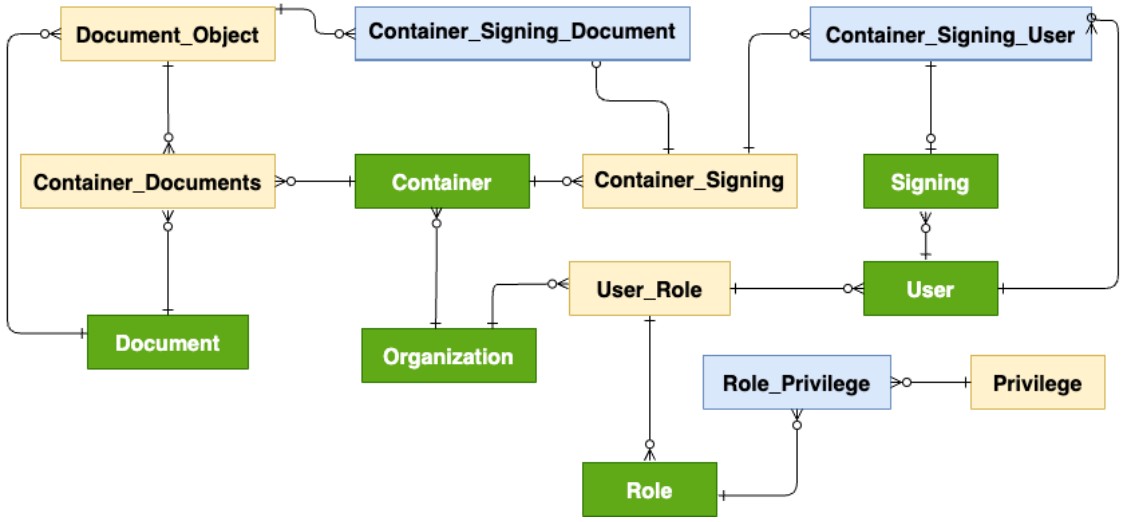
Käesoleva lõputöö skoobis ei looda täisfunktsionaalset eesrakendust koos disainitud kasutajaliidesega. Kuna kasutajaliidese loomine kuulub lõputöö konteksti ning on planeeritud järgmises skoobis, siis kirjeldatakse järgnevalt siiski ära loodav kasutajaliides. Arendatav toode realiseeritakse tarkvaraplatvormil Confluence Cloud toodetele *plugin*'i lahenduste *Forge Byline Item* ning *Forge Macros* arendamise kaudu. Ehitatavad *plugin*'id kuvatakse lõppkasutajale Confluence Cloud veebirakenduses ning ühe veebilehe osana neile eraldatud plokkis.

Seetõttu on oluline arvestada kogu ülejäänud Confluence Cloud kasutajaliidesega. Forge'i kasutajaliidese komplektis, ehk *UI kit's,* on loodud suurel hulgal komponente, mis järgib Atlassian'i toodete ülest disainisüsteemi [105]. Atlassian'i kasutusmugavuse spetsialistide ja disainerite poolt on arendajatele välja arendatud abistav disainisüsteem, mis järgib UX parimaid praktikaid ning mis tagab Atlassian'i platvormidele loodavate toodete ühtse kasutajaliidese teiste ülejäänud Atlassian'i tootelehega.

Käesoleva lõputöö konteksti loodava toote loomisel kasutatakse kasutajaliidese *Forge UI kit* komplektis pakutavaid komponente, mis on disainitud Atlassian'i disainisüsteemi järgides. Valitud lahendus aitab lõppkasutajal tagada hea kasutajakogemuse ning võimaldab kasutajal mugavalt kasutada lisatavad tooted. Samuti aitab valitud lahendus hoida kokku aega disaini arvelt, sest *Forge UI kit* komplektis on kõik vajalikud komponendid olemas, mida loodavas rakendus autor soovib kasutatakse.

3.9 Andmebaasi disain

Lähtuvalt kasutajalugudest koostati andmebaasi struktuurist ning olemitest. UML modelleerimiskeelt järgides on loodud loogiline olemi-suhte diagramm, ehk ERD[1], mis on joonise (Joonis 9) mahukuse tõttu on täiskujul leitav Lisa 8 alt. Joonisel 1 on kujutad kontseptuaalse olemi-suhte diagramm, kus roheline kujutab primaarseid, kollane sekundaarseid ning sinine siduvaid ja tehnilisi olemeid.



Joonis 1. Andmebaasi kontseptuaalse olemi-suhte diagramm

3.10 Analüüsi kokkuvõte

Nõuete määramisel on arvesse võttu, et on meditsiiniseadmete tootjate probleemi käsitletakse eelkõige Eesti kontekstis ning API-põhine tagarakendus peab olema äriliselt piisavalt tulus, et vähemalt katta jooksvad kulutused. Analüüsi etapis käsitleti erinevaid kasutajagruppe ning nendega funktsionaalseid ja mittefunktsionaalseid nõudeid.

Tehnoloogia poole pealt ilmnes analüüsi tulemusena, et tagarakendus ja selle äriloogika peaks eksisteerima veebiteenuse kujul, mis kasutab REST API põhimõtteid. Eraldiseisvalt eksisteerib veel kasutajapoolne klientliides, mis suhtleb teenusega läbi REST API päringute. Valitud lähenemine võimaldab tulevikus lihtsama vaevaga ehitada uusi liideseid ka teistele platvormidele, vajadusel vahetada välja nii kliendi- kui ka

[1] *Entity Relationship Diagram* – olemi-suhte diagramm, metoodika andmemudelite koostamiseks ja kirjelduse esitamiseks

serveripoolset raamistikku teineteist mõjutamata ning hoida andmeid lahus klientrakendusest.

Serveripoolsetest keeltest osutus valituks Node.js, millega on lõputöö autoril eelnevalt juba rohkelt kogemust ning millel on põhjalik tugi arendusvahendite, kompileerimistööriistade ja raamistike näol. Tagarakenduse raamistikuks valiti populaarne NestJS, mis on loodud Node.js tehnoloogiale. Tulevikus loodava kliendipoolse eesrakenduse ehitamisel kasutatakse Forge'i ja React'i tehnoloogiad, sest Atlassian'i tarkvaraplatvormile loodava *plugin*'i ehitamine on piiratud nimetatud tehnoloogiatega. Nii Tagarakenduse kui ka eesrakenduse koodi kirjutamisel kasutatakse TypeScript programmeerimiskeelt.

Serverilahenduseks on valitud *PaaS* lähenemine, kus serveripoolne haldus on jäetud teenusepakkuja hooleks ning mis võimaldab luua skaleeritavat tagarakendust.

Lahenduse koodibaasi arendatakse IntelliJ WebStrorm programmis ning andmebaasi lahenduseks on valitud avatud lähtekoodiga PostgreSQL. Koodi versioonikontrolli platvormiks on valitud Bitbucket, kuna autor omab tarkvara arendamisel varasemat kogemust Atlassiani tööriistadega, näiteks Confluence ning Jira.

4 Realisatsioon

Lõputöö kontekstis loodava lahenduse realisatsioon ja arendus on jaotatud nelja suuremasse peatükki: veebiteenus, klientrakendus, testimine ning pidev integratsioon. Tehnoloogilised üksikasjad ja aspektid, mis puudutavad nii teenusepoolset tagarakendust kui klientrakendust, on käsitletud mõlemad eraldi peatükkides.

4.1 Tagarakenduse lahendus

Tagarakenduse ehk *backend* on teenus, mis hoolitseb äriloogika ning andmete ligipääsu eest. Tagarakendus on ehitatud mikroteenuste arhitektuuri peale, mis koosneb iseseisvatest teenustest. Iga teenus on omakorda ehitatud NestJS raamistiku peale ning omab andmebaasi, kus on talletatud kõik äriloogikaga seonduv info. Teenused on internetis kättesaadavad REST API-liidese kaudu üle HTTPS protokolli. REST API päringute vastuvõtmise ja teenustesse suunamiste eest kannab hoolt sissepääsulüüsi teenus ehk *API-gateway* ning teenuste omavaheliseks suhtlemiseks kasutatakse gRPC [1]transpordikihti. Järgnevates peatükkides on lähemalt kirjeldatud tagarakenduse arenduse erinevaid aspekte ning tehnoloogilisi lahendusi.

4.1.1 Mikroteenuste arhitektuur

Tagarakendus on ehitatud mikroteenuste arhitektuuril, mis seob omavahel kokku kogumi teenustest üheks rakenduseks. Teenused suhtlevad omavahel üle võrgu, kasutades tehnoloogiliselt agnostilisi[2] protokolle nagu HTTP ja gRPC [106]. Mikroteenuste arhitektuuri loomisel on arvestatud järgmiste ühe tunduma mikroteenuste arhitektuuri eestvedaja Martin Fowler'i aastal 2014. avaldatud põhimõtetega [107]:

[1] Google *Remote Procedure Call* – Google poolt laiendus RPC kaugprotseduurikutse protokollile, mis võimaldab klientarvutil asuval programmil täita serverarvutil asuvat programmi

[2] Agnostiline – IT kontekstis viitab võimele, kui süsteem saab toimida teise süsteemi keskkonnas ilma, et see teaks või nõuaks midagi teise süsteemi üksikasjadest. Saavutatakse tavaliselt kokkulepitud standardite järgmise kaudu.

- Lõdvalt sidestatud (*Loosely decoupled*) – teenused on eraldiseisvad ning on sidestatud omavahel API liideste kaudu. Teenus ei pea teadma teise teenuse detaile ning ühes teenuses tehtav muudatus ei tohiks mõjutada teist teenust.
- Ühe vastutuse printsiip (*Single responsibility principle*) – iga teenus keskendub ainult selgele ärifunktsionaalsuse täitmisele ning järgib põhimõtet "Tee ühte asja, aga tee seda hästi."
- Kõrge sidusus (*High cohesion*) – teenuste sisemiste moodulite ja komponentide kood on loogiliselt organiseeritud ja kokku grupeeritud ning täitavad konkreetseid ülesandeid.

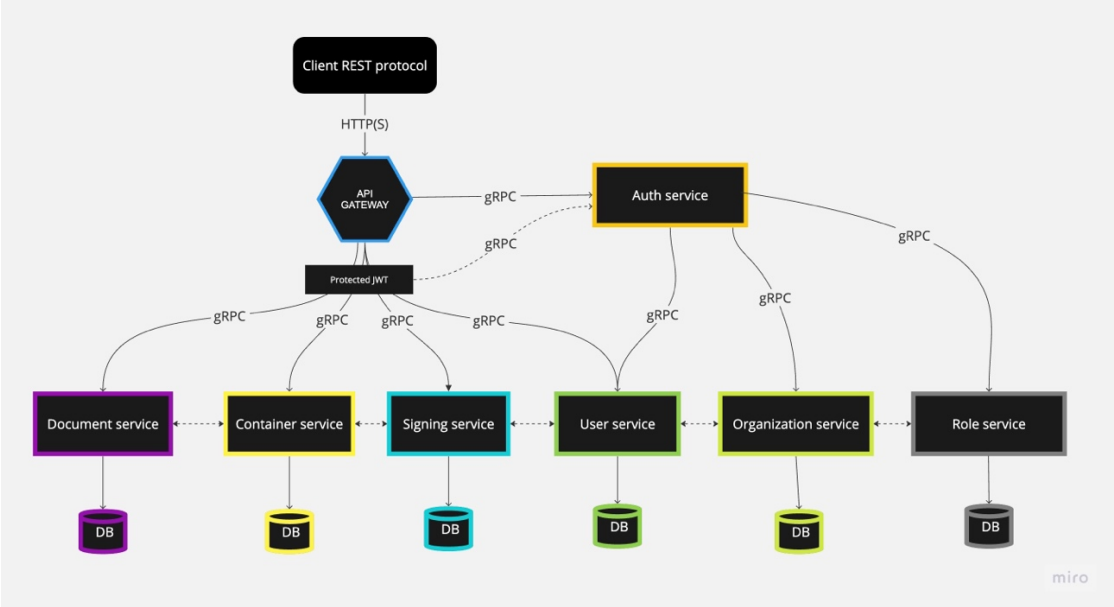
Mikroteenuste arhitektuuri koostamisel on lähtutud äriloogikast – iga teenus on piiritletud kindlate ärifunktsionaalsuste realiseerimisega. Arhitektuuriliselt on teenused tükeldatud vastutusala alusel järgmisteks loogilisteks teenusteks:

- Sissepääsulüüs ehk *API-gateway* – API-rakendusliidese võrgupunkt, mille kaudu toimub sissepääs teistesse rakenduse teenustesse. Sisse tulnud REST API päringud suunatakse edasi marsruutimise teel sobivasse teenusesse. Rakenduses on ainus teenus, mis on kättesaadav üle HTTP võrguprotokolli.
- Autentimine teenus, ehk *auth-service* – kasutaja autentsuse (identiteedi) kontrollimise ja tõestamise. Vastutab kasutajate ligipääsu JWT token'ite[1] loomise, dekodeerimise ja värskendamise eest. Kõik tehtavad võrgupäringud peavad olema autenditud ning sooritatud sisse loginud kasutajale määratud õiguste piires.
- Kasutajate teenus ehk *user-service* – vastutab kasutajakontode ning neile määratud rollide eest. Kirjete haldamine toimub CRUD-toimingude kaudu.
- Kasutajarollide ja privileegide teenus ehk *role-service* – kasutajarollide haldamise teenus. Kirjete haldamine toimub CRUD-toimingude kaudu.
- Allkirjastamise teenus ehk *signing-service* – digitaalse allkirja sooritamise teenus. Teenus on liidestatud Dokobit API-liidesega. Teenuses valmistatakse ette dokumendid allkirjastamiseks ning hallatakse allkirjastajaid.
- Dokumentide teenus ehk *document-service* – vastutab dokumentide haldamise eest. Teenus on liidestatud Confluence Cloud API-liidesega. Kirjete haldamine toimub CRUD-toimingude kaudu.

[1] Token – kasutaja autentimisel genereeritud turvamärk, mis manustatakse päringutele kaasa.

- Dokumentide konteinerite teenus ehk *container-service* – vastutab dokumentide konteinerite haldamise eest. Kirjete haldamine toimub CRUD-toimingude kaudu.

Järgneval joonisel (Joonis 2) on kujutatud tagarakenduste teenuste vahelist arhitektuuri.



Joonis 2. Rakenduse mikroteenuste arhitektuur

4.1.2 Konteinerite kiht

Loodavavaid mikroteenuseid käitatakse konteinerites. Konteiner on standardne tarkvarapakett, mis sisaldab ja pakendab kõik vajaliku ühe mikroteenuse käivitamiseks, näiteks programmikoodi, teeke, sõltuvusi, ja muid abivahendeid, millele vastav mikroteenus toetub. Standardsete konteinerite kasutamine võimaldab automatiseerida mikroteenuste käivitamist ja jooksutamist sõltumata aluseks oleva serveri operatsioonisüsteemist. Samuti on tänu konteineritele võimalik nõudluse kasvades mikroteenuste skaleerimist automatiseerida.

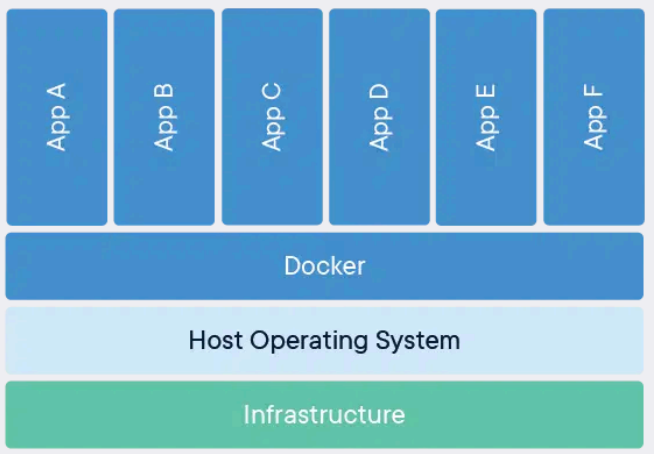
Konteinerite kasutamiseks on kasutuselevõetud Docker'i lahendus. Docker on Solomon Hykes'i poolt aastal 2013 arvutiprogramm, mis võimaldab operatsioonisüsteemi tasemel virtualiseerimist [1][108]. Docker teeb mugavaks ja võimalikult lihtsaks tarkvara arendamise ja serverisse paigutamise protsess, sest loodav tarkvara käitub sarnaselt

[1] Virtualiseerimine – operatsioonisüsteemikiht, mis võimaldab füüsilises arvutis kasutada erinevatel operatsioonisüsteemidel üksteisest isoleeritult ühiseid riistvararessursse.

erinevatel platvormidel. Docker'i avaliku tõmmiste register Docker Hub[1] pakub suurel hulgal eelkonfigureeritud konteinerite tõmmiseid[2], millede abil saab ehitada erinevaid rakendusi. Dockeri konteinerid sobivad hästi loodava mikroteenuste arhitektuuriga, võimaldades kasutada järgnevaid eeliseid tarkvara arendamisel ja serveris käitlemisel [109]:

- Automatiseerimise kiirendamine – arendamise elutsüklis on võimalik kõiki konteinerit eraldi serveris käsitelda ja konteinerite serverisse paigaldamiseks on saadaval mitmeid abivahendeid. Kogu protsess on tehtav skriptide abiga.
- Iseseisvuse suurendamine – konteinerid on eraldiseisvad üksused, mis võimaldavad süsteemi eri osasid arendada erinevates programmeerimiskeeltes ja erinevate tehnoloogiate abiga.
- Porditavus – Docker pakib konteinerisse kaasa kõik vajalikud sõltuvused, mis on saadaval igas keskkonnas kus konteinerit käivitada soovitakse.
- Vähene ressursside kasutus – Docker'i konteinerid sisaldavad endas ideaaljuhul vaid rakendust ning rakenduse sõltuvusi.

Joonisel 3 on visualiseeritud, kuidas Docker'i konteinerid kasutavad ühiseid ressursse.



Joonis 3. Docker'i konteinerite ühiste ressursside kasutamine

Konteiner põhiste rakenduste loomiseks kasutatakse Docker'i tehnoloogiat ning tõmmisfaili (*image*) ehitamise juhis koos konteineris käivitamise käskudega on

[1] https://hub.docker.com

[2] Tõmmis on fail, milles on salvestatud meediumi sisu täielik koopia koos struktuuriandmetega.

kirjeldatud failis *Dockerfile.* Mitmest konteinerist koosnevate tagarakenduse käitlemise mugandamise eesmärgil on kasutusele võetud Docker'i tööriist Docker Compose ning loodud *docker-compose.yml* konfiguratsioonifail, mis kasutab YAML[1] failiformaati. Docker Compose'i käivitamisel kasutatakse *docker-compose.yml* konfiguratsioonifaili ning selle alusel luuakse ja käivitatakse kõik kirjeldatud konteinerid. Rakenduses kasutatav *Dockerfile'i* koodinäide (Koodinäite 6) on leitav Lisa 9 alt ning osaline *docker-compose.yml* konfiguratsioonifaili koodinäide (Koodinäite 7) on leitav Lisa 10 alt.

4.1.3 Transpordi kiht

Loodaval RESTful rakendus on üle interneti kättesaadav üle turvalise HTTPS protokolli. Rakenduse sisemised teenused on välisvõrgule suletud läbi portide sulgemise. Ligipääs sisemistele teenustele toimub läbi sissepääsulüüsi teenuse ehk *API-gateway* kaudu [110].

Sisemised teenused on üksteise suhtes nähtavad ning kõik teenused saavad sooritada API päringuid teiste sisemiste teenuste poole. Tagarakenduse äriloogika on ära jaotatud erinevate omavahel seotud teenuste vahel ning iga eesrakendusest tehtud päringu teenindamiseks käivitatakse mitmeid teenuste vahelisi interaktsioone. See tõstab oluliselt võrgukoormust ning edastavaid andmemahte, mistõttu tavapärane võrguprotokoll HTTP ei ole kõige sobivam lahendus.

API-rakendusliidese on viimati mainitud põhjusel kasutusele võetud teenuste vaheliseks transpordikihiks gRPC [111]. gRPC on keele- ja platvormineutraalne tehnoloogia, mis loodi 2015 tehnoloogia ettevõtte Google initsiatiivil ning on vabavaraline laiendus 1970ndatel loodud RPC kaugprotseduurikutse protokollile. gRPC pakub pikalt kasutatud RPC protokollile värskendust, muutes selle koostalitlusvõimeliseks (*interoperability*), kaasaegseks ja tõhusaks, kasutades selliseid tehnoloogiaid nagu Protocol Buffers ja HTTP/2. Kõik see muudab andmete edastamise kiiremaks, väiksemaks ja lihtsamaks ning seetõttu skaleerub hästi loodava mikroteenuste arhitektuuriga [112].

Teenuste vaheliste andmete serialiseerimiseks[2] kasutakse Protocol Buffers (lühendina kasutatakse Protobuf) mehhanismi [113]. Protocol Buffers on keele- ja

[1] YAML Ain't Markup Language – inimsõbralik andmete serialiseerimise standard [137]

[2] Serialiseerimine - andmete teisendamine selliseks, et need oleks esitatavad järjest.

platvormineutraalne liidese määratluskeel, ehk IDL[1], mille kaudu defineeritakse andmestruktuuriskeemid ja programmeerimisliidesed. Struktureeritud liideste põhjal serialiseeritakse andmed enne edastamist klientteenusest masinloetavale binaarkujule ning vastuvõtvas serverteenuses deserialiseeritakse[2] andmed tagasi algsele inimloetavale struktureeritud kujule. Võrreldes JSON ja XML andmeformaatidega on Protobuf väiksema mahulisem ning andmete serialiseerimine on efektiivsem, sest andmeid edastatakse bitijadana ning andmete edastamine võib olla kuni 10 korda kiirem [114].

Protocol Buffers'i kasutamiseks on vajalik rakendusse paigaldada teekidena Protobuf definitsioonifailide kompilaatori, ehk *protoc* ning gRPC klientteenuse ja serverteenuse. Järgnvas koodinäites (Koodinäide 1) on rakenduse põhimooduli faili *main.ts,* milles on seadistatud gRPC server koos etteantud Protobuf definitsioonifailidega.

```
const path = join(__dirname, 'document-service', 'document.proto'),
const app = await NestFactory
      .createMicroservice<MicroserviceOptions>(AppModule, {
          transport: Transport.GRPC,
          options: {
            package: 'document',
            protoPath: path,
          },
       });
```

Koodinäide 1. gRPC seadistamine näide failis *main.ts*

Koodinäites 2 on leitav näide, kuidas mikroteenuses *signign-service* asuvas gRPC kontrollerfailis seotakse omavahel ära gRPC protseduur ning allkirjastamise teenuse meetod, kasutades selleks loodud abifunktsioone annotatsioonide kujul.

```
@GrpcMethod(EGrpcService.SIGNING, 'PrepareSigning')
async prepareSigning(
  @Payload() data: CreateSigningRequestDto,
  @GrpcUser() authorizedUser: IAuthorizedUser
): Promise<SigningResponseDto> {
     const signingData = await this.signinService.prepareSigning(data,
authorizedUser)
     return new SigningResponseDto(signingData);
}
```

Koodinäide 2. näide gRPC protseduuri ja teenuse sidumiseks

[1] IDL - Interface Definition Language

[2] Deserialiseerimine - andmete taastamine serialiseerimise eelseks seisundisse.

4.1.4 NestJS rakenduse arhitektuur

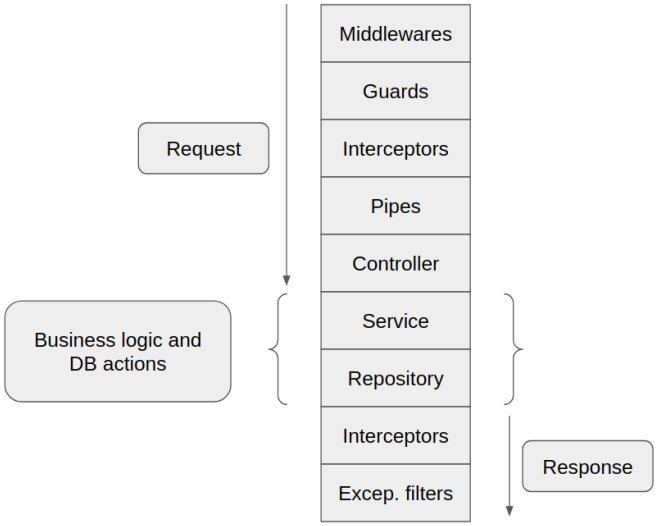
NestJS raamistik on sobilik keerukate tagarakenduste arendamiseks, kus on eemaldatud suur osa konfiguratsiooni ja selle asemel kasutatakse konventsioone annotatsioonide abil. NestJS raamistikuga kaasneb Express.js veebiserver, mis võimaldab ehitada REST API-liidest. Raamistikku on samuti loodud pakett mikroteenuste arhitektuuri ehitamiseks ning sisseehitatud tugi erinevate tarkvaraliste transpondrite[1] kasutamiseks, mis hoolitsevad sõnumite edastamise eest erinevate mikroteenuste instantside vahel.

Tagarakenduse teenused põhinevad kolmekihilisel arhitektuuril (*3-tier architecture*), kus erinevad arhitektuurised kihid on omavahel isoleeritud. NestJS raamistiku kihid on järgnevalt jaotatud [115], [116]:

1. Kontrollerid (*Controllers*) – kihi eesmärk on vastu võtta eesrakenduse päringud ja määrata REST API marsruudid (*route*)
2. Teenuste kiht (*Service Layer*) – sisaldab ainult rakenduse äriloogikat. Juhitakse kõiki CRUD-i toimingud ja meetodeid andmete töötlemiseks
3. Andmejuurdepääsu kiht (*Data Access Layer*) – see kiht määrab andmetele ligi pääsemise ja andmebaasis tehtavate toimingute viisid

Kihtide jaotamine kirjeldatud viisil võimaldab jagada rakendust paremini hallatavateks üksusteks vastutusala alusel. Kihtide üksteisest eraldamine lihtsustab muudatuste tegemist ning muudatusi saab teha ühe kihi piires või kaasata ainult lähimat kihti, ilma kogu programmi mõjutamata. NestJS raamistikus on 3-kihiline arhitektuur omakorda jaotatud veel väiksemateks loogilisteks kihtideks. Järgmisel joonisel (Joonisel 4) on visuaalselt esitatud NestJS raamistikkus kasutatavad kihid rakendamise järjekorras. Tuvastatud vea või esinenud tõrkede korral jäetakse vahele järgnevad kihid ning saadetakse päring kohaselt edasi erandite haldamisse kihti, ehk *Expection filters.*

[1] Transponder (*transporter*) – tehnoloogiakiht, mis vastutavad sõnumite edastamise eest erinevate süsteemide vahel.



Joonis 4. NestJS 3-kihiline arhitektuur

NestJS pakub samuti suurel hulgal erinevaid CLI käskudega käivitatavaid tööriistu, mis teevad projekti ehitamise, arendamise ja teekide haldamise mugavaks. [109]. Joonisel 5 on välja toodud võimalikud NestJS raamistikku käsurealt käivitatavad käsud rakenduse ehitamiseks või laiendamiseks:

```
Commands:
  new|n [options] [name]                          Generate Nest application.
  build [options] [app]                           Build Nest application.
  start [options] [app]                           Run Nest application.
  info|i                                          Display Nest project details.
  add [options] <library>                         Adds support for an external library to your project.
  generate|g [options] <schematic> [name] [path]  Generate a Nest element.
    Schematics available on @nestjs/schematics collection:

    name            alias        description
    application     application  Generate a new application workspace
    class           cl           Generate a new class
    configuration   config       Generate a CLI configuration file
    controller      co           Generate a controller declaration
    decorator       d            Generate a custom decorator
    filter          f            Generate a filter declaration
    gateway         ga           Generate a gateway declaration
    guard           gu           Generate a guard declaration
    interceptor     itc          Generate an interceptor declaration
    interface       itf          Generate an interface
    middleware      mi           Generate a middleware declaration
    module          mo           Generate a module declaration
    pipe            pi           Generate a pipe declaration
    provider        pr           Generate a provider declaration
    resolver        r            Generate a GraphQL resolver declaration
    service         s            Generate a service declaration
    library         lib          Generate a new library within a monorepo
    sub-app         app          Generate a new application within a monorepo
    resource        res          Generate a new CRUD resource
```

Joonis 5. NestJS käsurealt käivatavad käsud rakenduse ehitamiseks

Käesoleva projekti ehitamise ja teekide haldamise tööriistaks valiti paketihaldur NPM versiooniga 8.12, JavaScript'i transpileriks[1] valiti TypeScript versiooniga 4.9.4 ning

[1] Transpiler - programm, mis teisendab ühe kõrgema taseme keele sisendkood teise kõrgema taseme keele väljundkoodiks

installeeriti lisateegid. Teekide valik on antud eelkõige mugavuse tõttu, kuna teeke on võimalik kergelt projekti lisada ka peale projekti loomist.

Loodud baasprojekt (*boilerplate*) sisaldab vaikimisi vaid baasklasse ja teeke rakenduse käitlemiseks ning arendamiseks. Samuti sisaldab projekti *metadata*[1] faili *nest-cli.json* ja paketihalduri konfiguratsioonifaili *package.json*, kus on loetletud projekti teegid, käivitavad käsurea skriptid ning paketi *metadata*.

Projekti loodavate klasside haldamiseks on arendatavas rakenduses jaotatud failid järgnevatesse loogilistesse kaustadesse, millele eraldiseisvalt lisanduvad rakenduse tuumiku failid *app.module.ts ja main.ts*, projekti *metadata* fail *nest-cli.json*, paketihalduri konfiguratsioonifail *package.json* ning Typescript'i konfiguratsioonifail *tsconfig.json*:

- *constants* – sisaldab kõiki muutumatuid väärtusi ja nimetusi, mida kasutatakse esmaste andmete genereerimiseks ja edasiseks töötlemiseks teenustes.
- *services* - sisaldab loodavaid teenuseid, mis loovad teenuste kihi. Teenuste, ehk *providers*, käivitamine ja haldamine toimub rakenduse Sõltuvuste süstimise[2] (*Dependency Injection)* halduri kaudu.
- *controllers* – REST API kontrollerid.
- *entities* – sisaldab ORM andmebaasi olemite mudeleid, mis võimaldavad teistel klassidel loetaval viisil pärida infot andmebaasist.
- *filters* – sisaldab kasutatavaid vahevaraseid erandite ja tõrgete kinni püüdmiseks
- *middlewares* – sisaldab kasutatavaid vahevaraseid, mis asetsevad teenustes sissetuleva päringu ning kontrollerite vahel.
- *dtos* – sisaldab sissetulevate päringute andmete valideerimisel kasutavaid andmeedastusobjekte, ehk DTO'd[3]. Samuti määrarakse väljaminevate andmete struktuur. Klassid, mis väljendavad domeeniobjekte äriloogika perspektiivist.
- *interfaces* – rakenduses kasutatavad liidesed.
- *enums* –kasutatavad andmetüübid, mis sisaldab kindlat konstantide komplekti

[1] Metaandmed. Andmed, mis kirjeldavad teisi andmeid

[2] Sõltuvuste süstimise haldur on rakenduse arhitektuuriline disainimuster, mis näeb ette objektide loomise eraldamist programmi käitumisest. Antud lähenemisel ei loo klassid mitte ise vajalikke objekte, vaid need antakse ette väljastpoolt.

[3] DTO ehk Data Transfer Object

- *interceptors* – Aspektorienteeritud programmeerimisest[1] inspireeritud korduvkasutavad klassid, mis hõlbustavad põhirakenduse loogika ja tavaliste korratavate ülesannete eraldamist. Kasutatakse sisendi valideerimisel, autentimisel, turvalisuse loogikaga lisamisel, vigade käsitlemise ja logimisel.
- *decorators* – annotatsiooni klassid, millega on võimalik rakendada kontrolli meetoditele täiendavad funktsionaalsust.
- *utils* – sisaldab abifunktsioone, mida kasutatakse erinevates teenustes.

Eelnevalt loetletud kaustadesse jaotatud klasse on NestJS raamistikkus mugav kasutada, sest raamistikku on sisse ehitatud kihiline elutsükkel (*lifecycle*) ning Sõltuvuste süstimise haldur (*Dependency Injection)*, mis võimaldab kihtide vahele lisada täiendatavat funktsionaalsust ja andmetöötlust.

4.1.5 Andmebaasi kiht

Node.js keelele on loodud rohkelt abitehnoloogiad, mis võimaldavad paindlikult rakendusi integreerida soovitud andmebaasi tehnoloogiatega [118]. NestJS raamistikku jaoks on veel loodud täiendavaid pakette, mis võimaldavad mugavalt andmebaasi integratsiooni hallata Sõltuvuste süstimise (*Dependency Injection)* halduri kaudu. [111] NestJS on andmebaasi agnostiline, mis võimaldab hõlpsasti integreerida suurema osaga relatsiooniliste või mitte- relatsioonilise andmebaasiga, kaasa arvatud PostgreSQL.

PostgreSQL on relatsioonilise andmebaasi, mille puhul NestJS raamistik soovitab kasutada populaarset TypeORM paketi. TypeORM on ORM (*object-relational mapping*) lahendus, mis pakub klasse ja meetodeid andmebaasiga suhtlemiseks ning võimaldab objekt-orienteeritud domeenimudelite ühendamist andmebaasiga. PostgreSQL'i andmebaasiga ühildumiseks tuleb installeerida PostgreSQL'i draiveri ning TypeORM vajalikud teegid.

Andmebaasiga ühenduse seadistamiseks pakub TypeORM mitmeid erinevaid võimalusi. Staatilise konfiguratsiooni lisamiseks ja dialekti määramiseks on võimalik lisada rakenduse põhimooduli faili *app.module.ts* vastavad andmebaasi parameetrid või luua

[1] Aspektorienteeritud programmeerimine – programmeerise paradigma, kus eraldatakse põhirakenduse loogika ja tavaliselt korratavad ülesanded

rakenduse juurkataloogi uus konfigiratsioonifail *ormconfig.json*, kus struktureeritult on kirjeldatud parameetrid andmebaasiga ühendamiseks. Mõlemad lähenemised on problemaatilised, sest kood koos andmebaasi ühendamise parameetritega jõuavad versioonikontrolli. Samuti ei ole võimalik seadistada parameetrid rakenduse käivitamise keskkonna põhiselt. Parem lahendus on kasutada keskkonnamuutujaid andmebaasi parameetrite määramiseks. TypeORM paketis on etteantud suurel hulgal keskkonnamuutujate võtmeid, mida on võimalik kasutada parameetrite määramiseks.

TypeORM pakub võimalust sünkroniseerida andmebaasi automaatselt domeeniobjektide alusel. Omadus on kasulik, et kiiresti testida loodud olemeid ning luua teste funktsionaalsusi katsetamisteks. Reaalses rakenduses keskkonnas (*production environment*) on automaatne andmebaasi sünkroniseerime ohtlik, kuna automaatsed andmebaasimuudatused on väga tõenäolised ning mitmes erinevas keskkonnas jooksva rakenduse andmebaase on ilma loogilise korrata raske hallata. TypeORM pakub sel põhjusel sisse ehitatud andmebaasi versioonihaldust läbi migratsioonide [120]. Tööpõhimõte on teha andmebaasi migratsioonid ära enne rakenduse käivitamist projekti lisatud migratsioonifaile abil, mida TypeORM failide asukoha ning nimede alusel.

TypeORM migratsioone ei oska kasutada teised Node.js ORM'd ning ORM abitehnoloogia vahetamisel on migratsioonid vaja uuesti luua. Viimasel põhjusel on rakenduses migratsioonide loomiseks kasutatud eraldiseisvat Knex.js andmebaasi ühildamise abitehnoloogia. Knex.js'i migratsioonide tööriist pakub andmebaasi versioonihalduste lahendust, mis ei sõltu NestJS rakenduses tehtud ORM valikust ning mille tööpõhimõtte ja kaustastruktuur on sarnane TypeORM migratsioonidega [121].

Lisa 11 all on esitatud näide (Koodinäide 8) migratsioonifailist rolli teenuse andmebaasist, mille kaudu luuakse rollide olemite tabelid. Migratsioonide mugavamaks käivitamiseks on kirjeldatud failis *package.json* täiendavad käsurealt käivitavad abiskriptid. Järgnevas tabelis (Tabel 11) on välja toodud täiendavad migratsiooni abiskriptid.

Tabel 11. Andmebaasi migratsioonide haldamise abiskriptid

Käsurealt abiskripti käsk	**Abiskripti kirjeldus**
npm run db:migrate:make \ --service={teenus} \ --migration_name={faili nimetus}	Genereeritakse määratud teenuses uus migratsiooni fail etteantud nimetusega. Faili nimesse lisatakse sufiksina (*suffix*) ajatempel.
npm run db:migrate:latest --service={teenus}	Määratud teenuses leitakse veel käivitamata failid ja migreeritakse andmebaasi muudatused
npm run db:migrate:rollback --service={teenus}	Määratud teenuses viimaste muudatuste tagasi tõmbamine.
npm run db:seed:make --service={teenus}	Määratud teenuses genereeritakse andmete külvamise fail
npm run db:seed:latest --service={teenus}	Määratud teenuses käivitakse kõik leitud andmete külvamise failid

Lähtuvalt mikroteenuste arhitektuurist on igal teenusel eraldiseisev andmebaas ning seetõttu Lisas 8. kujutletud UML modelleerimiskeele baasil loodud andmebaas loogilise olemi-suhte diagrammi jaotatakse veel mitmeks väiksemaks eraldiseisvaks andmebaasiks. Sel põhjusel ei ole võimalik realiseerida kõiki seoseid läbi välisvõtme *(Foreign key*[1]), sest kõigi tabelite vahel ei pruugi olla füüsilist ühendust.

Tagarakenduse andmebaasides on primaatvõtmete (*Primary key*[2]) talletamiseks kasutatutud UUID[3] andmetüüpi, mille väärtus genereeritakse automaatselt andmebaasi poolt kasutades PostgreSQL funktsiooni *uuid_generate_v4()* [122], [123]. Primaatvõtmetel UUID'e (versioon 4.) väärtuste kasutamine garanteerib praktikas unikaalsed primaarvõtme väärtused üle kõigi teenuste andmetabelite [124]. Samuti ei ole

[1] Välisvõti (*Foreign key*) – relatsioonilises andmebaasis olev andmeveerg, mille kaudu seostatakse olemite omavahelist seost [140]

[2] Primaarvõti (*Primary key*) – andmeveerg, mis on valitud kirjet unikaalselt identifitseerima [140]

[3] UUID (*Universally Unique Identifier*) – universaalselt ainulaadne identifikaator

võimalik UUID põhjal tuletada äriliselt tundlikku informatsiooni, näiteks kui palju kirjeid on andmetabelisse loodud.

Rakenduse arenduse esimeseks sammuks on migratsiooni failide loomine ning vastavate abiskriptide käivitamine käsurealt. Teise sammuna järgneb domeeniobjektide loomine eelnevalt disainitud olem-suhte diagrammi alusel ning loodud klasside registreerime *app.module.ts* põhimooduli failis. Annotatsiooni *@Entity* ja *@Column* lisamine on vajalik domeeniobjektide klassides, et TypeORM oleks teadlik tõlgendatavatest objektidest.

4.1.6 REST API

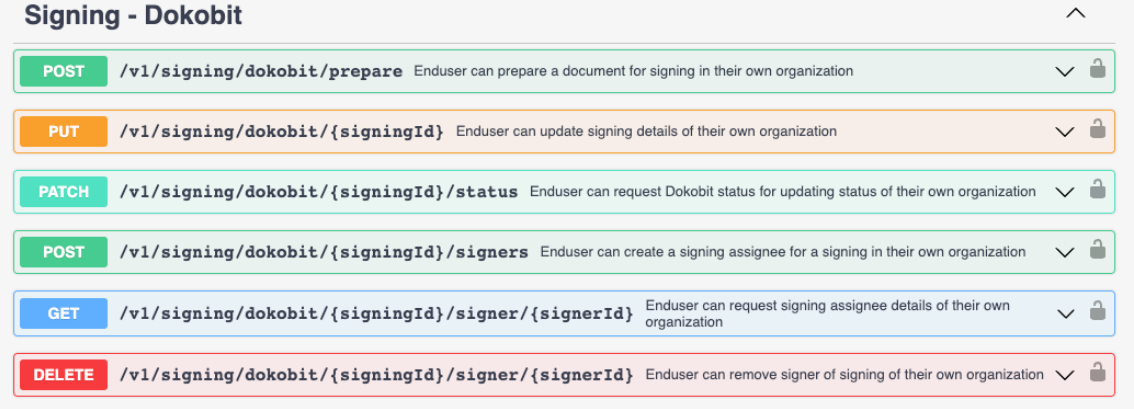
Loodud REST API teekondade disainimisel on aluseks võetud kasutajalood ning lõputöö kontekstis loodav kasutajaliides. Rakenduses on REST API ressursside kirjeldamisel järgitud üldlevinud tava ning URL-d algavad */api* sufiksiga, millele järgneb käesoleva ressursi versioon [125]. Rakendus tagastab iga päringuga vastava tagastuskoodi (*status code*) vastavalt kokkulepitud tavale, mis väljendavad päringu võimalikke tulemusi. Käesolevas rakenduses kõige sageli esinevad tagastuskoodid on 200, 201, 203, 400, 401, 403, 404, 412 ning 500 [126].

Päringud on jaotatud kontrollerite põhjal erinevatesse sektsioonidesse, kus on täpsustud päringu kirjeldus, tüüp (GET, POST, PUT või DELETE) ning ressurss. Iga päringu ette tuleb lisada ka */api/v1,* mis tähistab, et tegemist on API ressursiga, mille versioon on üks. Päringuga võib kaasneda lisainfot või andmeid, mida on võimalik edastada kas päringu parameetrina (*query params*), mis asuvad päringu URL-ile lisatuna või JSON objektina, mis asub päringu *Body*'s.

Kontrollerfailides on kirjeldatud iga päringu kohta andmed annotatsioonide abil. Kirjelduste põhjal genereeritakse automaatselt OpenAPI struktuurile vastav REST API OAS2[1] spetsifikatsioonifail [127]. Loodud OAS2 spetsifikatsioonifaili kasutab REST API esitlemise tööriist Swagger, et luua rakendusele interaktiivne REST API elav-dokumentatsioon (*live documentation*) HTML lehekujul [128]. Joonisel 6 on nähtav

[1] OAS2 (*OpenAPI Specification*) – OpenAPI spetsifikatsioon versiooniga 2

osaline näide Swagger'i poolt genereeritud API kirjeldus juurteekonnale *api/v1/signing/dokobit*.



Joonis 6. Swagger'i dokumentatsioonist osaline näide juurteekonnale *api/v1/signing/dokobit*.

Lisas 12 leitavas koodinäites (Koodinäide 9) on esitatud osa REST API kontrollerfailist *document-confluence.controller.ts*, milles on API-teekonnale (*API-endpoint*) POST *api/v1/documents/confluence/{documentId}/objects* lisatud kirjeldused OpenAPI dokumenteerimiseks annotatsioonide kaudu.

Rakenduses on loodud mitmeid REST API-teekondi, mis kirjeldavad teenuse funktsionaalsusi. Lisas 18 on leitav juhend rakenduse API dokumentatsiooni avamiseks. Välja on jäetud päringud, mis ei kuulu praegusesse skoopi.

4.1.7 Dokobit liidestus

Dokumentide digitaalseks allkirjastamiseks on kasutatud Dokobit'i kolmanda osapoole teenust. [129] Dokobit pakub mitmeid robustseid lahendusi dokumentide allkirjastamiseks, mis võimaldavad tarkvaraplatvormil Confluence Cloud sooritada dokumentide digitaalset allkirjastamist. Lõputöös loodavas lahenduses luuakse dokumentide allkirjastamiseks eraldiseisev teenus, mis liidestakse Dokobit'i teenusega *Documents Gateway*[1].

Dokumentide allkirjastamine protsess hõlmab endast mitmeid tagarakenduste vahelisi tegevusi kui ka kasutajaliideses tehtavaid toiminguid, mida sooritab lõppkasutaja.

[1] Dokobit Gateway API dokumentatsioon - https://gateway-sandbox.dokobit.com/api/doc

Tagarakenduses sooritakse allkirjastatud dokumenti ettevalmistamiseks mitmeid tegevusi. Mitmeosaline protsess on järgnevalt kirjeldatakse:

- Faili ettevalmistamine – esmalt laetakse allkirjastatav Confluence Cloud sisuleht tagarakendus puhvrisse kasutades Confluence Cloud API teenust sisulehe eksportimiseks. Sisuleht eksporditakse PDF failiformaati.
- Faili üleslaadimine – Dokobit'i süsteemi lisatakse allkirjastav dokument. Dokobit'i API'sse üleslaetav fail peavad olema kodeeritud Base64[1] kodeerimismeetodiga.
- Allkirjastajate lisamine – Kasutajaliidese kaudu lisatud allkirjastajad lisatakse Dokobit'i süsteemi kui dokumenti allkirjastajad. Tagarakenduses ei salvestada kasutajate isikukoodi ega muid kontaktandmeid ning seetõttu peab dokumenti ettevalmistav lõppkasutaja need väärtused manuaalselt täitma.
- Allkirjastamine – allkirjastamiseks kasutatakse Dokobit Gateway kasutajaliidest, mis kuvatakse Confluence Cloud sisulehel dialoogakna sees.
- Allkirjastatud dokumenti allalaadimine – peale kõigi osapoolte edukat dokumenti allkirjastamisest lisatakse allkirjastatud dokument Confluence Cloud sisulehe manusess.

Dokumenti allkirjastamiseks ettevalmistamise protsessist parema ülevaade andmiseks on Lisa 17 alt leitav järgnevusdiagramm (Joonis 10), mis kujutab kuidas informatsioon täpselt liigub ning milliseid sündmusi süsteemis sooritatakse.

4.1.8 API-rakendusliidese turvalisus

Autentimine on iga rakenduse üks olulisemaid aspekte. See parandab rakenduse turvalisust, kontrollides kasutajaid enne juurdepääsu andmist rakenduse eri osadele. Autentimine on kasutaja või seadme kontrollimise protsess enne süsteemile või ressurssidele juurdepääsu võimaldamist. [130].

NestJS pakub nii sisse ehitatud kui ka väliseid teeke, mis mitmeid turvalisust tagavaid kihte ja strateegiad. NestJS turvalisuse kiht kujutab endas filtrit, mis asetseb teenuses sissetuleva päringu ning kontrollerite vahel. Filtreid võib olla mitu ning nendel on

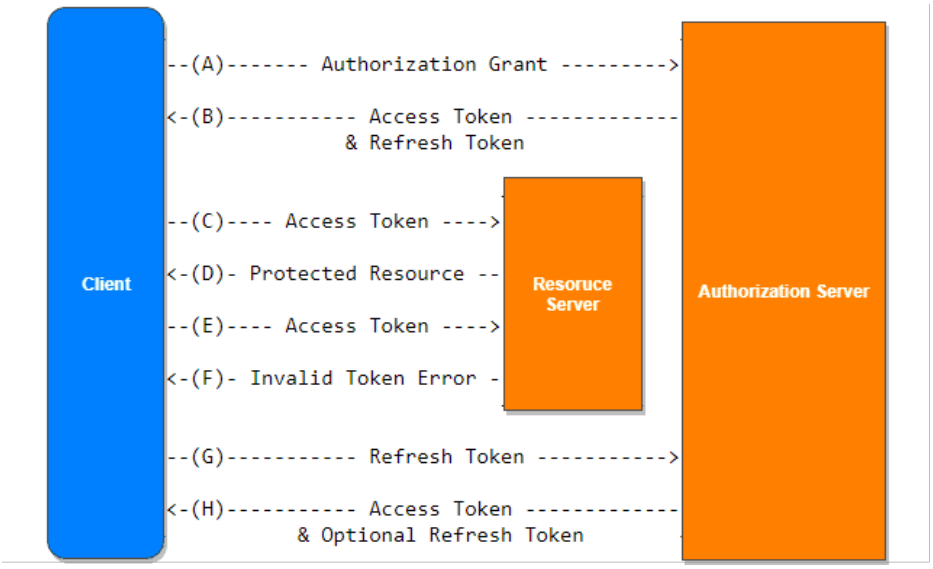
[1] Base64 - kodeerimismeetod, mis teisendab binaarandmeid ASCII tekstiks ja vastupidi.

omavahel hierarhia. Kui päring ei pääse läbi esimese filtri, ei saadeta seda enam järgnevatele filtritele.

Turvalisuse eesmärgil on tagarakenduses kasutusele võetud levinud *JSON Web Token*, ehk JWT, standard, mis avatud standardi numbriga RFC 7519 [131]. JWT määratleb kompaktse ja iseseisva viisi osapooltevahelise teabe turvaliseks edastamiseks JSON formaadis objektina. Edastatud teavet saab kontrollida ja usaldada, kuna see on digitaalselt signeeritud. JWT luuakse ja tagastatakse tagarakenduses poolt peale kasutaja autentimist ning edastatakse kaasa kõigi järgnevate päringutega, mis nõuavad autenditud kasutajat.

JWT koosneb kolmest komponendist: *header* ehk päis, *payload* ehk kasutaja kohta käiv informatsioon ning *signature* ehk allkiri. Päis määrab ära *token*'i ehk sõne tüübi ning sisu kaitsmiseks kasutatavad krüptoalgoritmi. *Payload*'is sisaldab kontrollitavaid turbeavaldusi, nagu kasutaja identiteet ja neile lubatud õigused.

NestJS tagarakenduses kasutatakse on SHA-265 kasutusel krüptoalgoritmina. Lühiajalise JWT (*Access token*) kehtivusajaks on määratud 15 minutit ning pikaajalise JWT (*Refresh token*) kehtivusajaks on määratud 1 tund. Turvalisuse tagamiseks tuleb JWT'd saata üle turvalise kanali nagu HTTPS. Järgneval joonisel (Joonis 7) on visualiseeritud protsess, kuidas toimub JWT token'ite omandamine:



Joonis 7. JWT token'ite omandamise protsess

Kõigil rakenduses defineeritud API päringutel on oma kasutajaõiguste tasand, mis määratakse kontrollerites kasutades annotatsiooni klassi *Permission guard*. Rakenduses kasutatavad kasutajaõiguste tasemed on selgitatud eelmisel analüüsi peatükis. Valvur (*guard*) võimaldab tagada, et kõik kasutaja tegevused sooritatakse vastavalt nende ligipääsu tasandile.

4.1.9 Konfiguratsioon

Loodav rakendus töötab erinevates keskkondades ning seetõttu tuleb kasutada keskkonna põhiseid konfiguratsioonisätteid. Iga keskkonna jaoks on oma määratud komplekt konfiguratsioonisätteid. Kuna konfiguratsioonisättede väärtused muutuvad igas keskkonnas, siis parimaks tavaks konfiguratsioonimuutujate salvestamiseks on keskkond ise [132].

NestJS võimaldab seadistada keskkonnamuutujate kasutamist läbi sisse ehitatud konfiguratsiooniteenuse. Rakendus oskab leida keskkonnast etteantud keskkonnamuutujad ning teisendab need vajadusel sobivale kujule. Samuti otsib NestJS rakenduse juurkataloogist konfiguratsioonifaili nimetusega *.env* ning faili leidmisel loeb rakendus määratud väärtused mällu. Rakenduse arendamiseks ja testimiseks luuakse lokaalne *.env.local* fail, mis sisaldab kõiki kasutavaid keskkonnamuutujaid. Konfiguratsiooniteenuse registreerimine põhimooduli failis *app.module.ts* on kujutatud järgneval koodinäites (Koodinäide 3):

```
ConfigModule.forRoot({
  isGlobal: true,
  load: [Configuration],
}),
```

Koodinäide 3. Konfiguratsiooniteenuse registreerimine põhimoodulis.

Mugavamaks keskkonnamuutujate kasutamiseks rakenduses on loodud täiendav abistav konfiguratsioonifail. Täiendavas konfiguratsioonifailis on määratud vaikeväärtused ning võtmepaarid, mille kaudu tehakse keskkonnamuutujad rakenduses kättesaadavaks. Abistava konfiguratsioonifaili *configuration.ts* koodinäide (Koodinäite X) on esitatud Lisas 13.

Konfiguratsioonifailis on täpsustatud nii *production*, *staging* kui ka arenduskeskkonna sätted. Samuti sätted andmebaasiga ühendamiseks, CORS[1] seadistused ning pordinumber, mille kaudu on rakendus keskkonnas ligipääsetav.

4.1.10 Valideerimine

Hea tava on kontrollida ning valideerida eesrakendusest saadetud andmete korrektsust. Sissetulevate päringute automaatseks valideerimiseks pakub NestJS raamistik mitmeid kaasatulnud võimalusi. Valideerimise eest hoolitseb rakenduses eraldi kiht, ehk *ValidationPipe*. Andmete automaatseks valideerimiseks tuleb määrada API-teekonnale valideerimisklass [133]. Valideerimisklass defineeritakse kui andmeedastusobjekti, ehk DTO klassina. DTO klassis kasutatakse andmereeglite ja struktuuri defineerimisel annotatsioone, mis tulenevad Typescript'i teeki *class-validator* spetsifikatsioonist. Lisas 14 on leitav koodinäide (Koodinäite X), mis esitleb valideerimisklassi näidet *create-document-request.dto.ts* põhjal.

4.1.11 Muud kasutatud tehnoloogiad

Järgnevalt on kirjeldatud teenusepoolseid tehnoloogiaid, mida eelnevad peatükid ei katnud:

- Webpack – laialt levinud Javascript'i teek, mis lubab mugavalt lähtekoodi mooduleid komplekteerida.
- Eslint – laialt levinud koodi valideerimise tööriist lähtekoodile.
- Prettier – laialt levinud koodivormindaja.
- Abiskriptid – rakenduse *tools* kaustas asuvad Bash'i abiskriptid andmebaasi ning rakenduse ehitamise mugavamaks kasutamiseks arenduses.
- Postman – tööriist, mille kaudu käivitada ja testida API kutseid.

4.2 Klientrakendusepoolne lahendus

Käesoleva lõputöö skoobis ei looda täisfunktsionaalset eesrakendust koos disainitud kasutajaliidesega. Kuna kasutajaliidese loomine kuulub lõputöö konteksti ning on planeeritud järgmises skoobis, siis kirjeldatakse järgnevalt ära tulevikus loodava

[1] *Cross-origin resource sharing* - mehhanism, mis lubab domeenivälist infovahetust [136]

eesrakenduse lahendus. Eesrakenduse varajane installeerimine ja eelseadistamine võimaldab varakult valideerida ja täpsustada nõudeid eesrakendusele.

Eesrakendus ehk kliendipoolne rakendus on eraldiseisev tagarakendusest ning vastavalt analüüsile kasutatakse Atlassian Forge raamistikku, mis võimaldab Confluence Cloud'i toode instantsidele lisada täiendavat funktsionaalsust. Atlassian Forge on ehitatud React raamistikku peale, mis kujutab endas JavaScript'i pakettide kogumit.

Tulevikus arendatav toode realiseeritakse tarkvaraplatvormil Confluence Cloud *plugin*'ite kujul ning arendamisel kasutatakse võimalikke lisandmooduleid *Forge Byline Item* ning *Forge Macros*. Confluence Cloud'i veebirakenduses käivitatakse installeeritud *plugin*'id ning kasutajaliideses kuvatakse neid ühe osana selleks eraldatud lehe plokkis. Sel põhjusel ei ole lõputöö konteksti loodavat eesrakendust vajalik ehitada kui täisfunktsionaalset eesrakendust. Confluence Cloud eesrakendusega kaasnevad juba kõik eesrakenduse funktsionaalsused ning samuti komplekt kasutajaliidese komponente.

Tulevikus kasutatakse loodava eesrakenduse loomisel *Forge UI kit* komplektis kasutavaid komponente, mis on kujundatud Atlassian'i disainisüsteemi järgides. *Forge UI kit* komplektis on olemas kõik vajalikud kasutajaliidesekomponendid, mida loodavas eesrakenduses autor plaanib kasutada. Eesrakenduses on kavas veel täiendavaid ehitada uusi korduvkasutatavaid ja kergesti hallatavaid komponente, mis kasutavad juba olemasolevaid *Forge UI kit* komplektis pakutavaid komponente.

Lõputöös skoobis käsitletav projekt ehitatakse Forge CLI käsurea komplekti kaudu, mis on Atlassian'i poolt pakutav viis luua Forge baasprojekti. Projekti loomiseks on vaja paketihaldurit NPM'i ehk Node Package Manageri, mille abil hallatakse teeke ning abiskripte projekti arendamiseks.

4.2.1 Forge projekti loomine

Forge projekti loomiseks tuleb esmalt luua uus Confluence Cloud rakendus ehk Forge App. Loodava Forge App'i projekti kaudu arendatakse kõik toode jaoks vajalikud kasutajaliidese lisandmoodulid ning hallatakse paigaldamise seadeid.

Forge App projekti loomiseks on vaja kasutada Forge CLI käsurea komplekti ning käivitada käsurea käsk *forge create*. Käsu käivitades avaneb graafiline käsurea kasutajaliides, milles tuleb teha valikud soovitud projekti ehitamiseks.

Peale valikute tegemist ehitatakse projekti jaoks uus baasrakendus ning luuakse projekti failid koos kaustastruktuuriga. Samuti registreeritakse uus rakendus Atlassian'i süsteemis ning ehitatakse Atlassian'i infrastruktuuris rakenduse jaoks 3 uut keskkonda rakenduse arendamiseks, testimiseks ning toode avaldamiseks.

Projekti konfiguratsioon määratakse *manifest.yml* failis, mis kasutab YAML[1] failiformaati. YAML on levinud *markup* keel, mida kasutatakse sageli konfiguratsioonifailide loomiseks, kuna on kergesti loetavad ning ühilduvad enamiku programmeerimiskeeltega. Loodud konfiguratsioonifail on välja toodud järgnevas koodinäites (Koodinäide 4).

```
modules:
  confluence:contentBylineItem:
    - key: document-signing-confluence-app-hello-world
      function: main
      title: Document-signing-confluence-app
      tooltip: Hello tooltip
      description: Displays Hello world!
  function:
    - key: main
      handler: index.run
app:
  id: '<application id>'
```

Koodinäide 4. Forge rakenduse konfiguratsioonifail manifest.yml

4.2.2 Forge Page Byline

Tarkvaraplatvormi Confluence Cloud'i *plugin Forge Byline Item* võimaldab sisulehtede tegevuste ribale lisada täiendavad funktsionaalsust ning elemente. Samuti võimaldab *plugin* luua avatavaid dialoogkaste, mis võimaldab mugavalt kuvada mahukamat sisu.

Tulevikus loodavas eesrakenduses kasutatakse *Forge Byline Item plugin*'it, et sooritada allkirjastamisega seotud toiminguid ning avada dialoogkastis Dokobit' allkirjastamise kuva. Lisaks võimaldab *plugin* kuvada sisulehe allkirjastamise olekut. *Forge Byline Item* lisandmoodul on liidestatud tagarakendusega ning vajalikku andmete kuvamisega tuleb sooritada erinevad REST API päringuid tagarakenduse poole.

[1] YAML Ain't Markup Language – inimsõbralik andmete serialiseerimise standard [137]

4.2.3 Forge Macros

Tarkvaraplatvormi Confluence Cloud *plugin Forge Macros* võimaldab sisulehtede sisuosa vahel käivitada pistikprogramme, mis võimaldab lisada täiendavad funktsionaalsust ning elemente.

Lõputöö konteksti kasutatava *Forge Macros plugin'*i funktsionaalsus on kuvada sisulehe sisuosa vahele allkirjastamine staatus ning kõiki allkirjastajate andmed nimekirjana. *Forge Macros plugin* on liidestatud tagarakendusega ning vajalikku andmete kuvamisega sooritatakse erinevad REST API päringuid tagarakenduse poole.

4.2.4 Suhtlus tagarakendusega

Loodavas eesrakenduses kasutatakse tagarakendusega suhtlemiseks Tanstack Query teeki [134]. Tanstack Query, ehk React Query, on populaarne teek andme päringute tegemiseks React'i programmides. React Query lihtsustab veebirakendustes andmete pärimist, vahemällu salvestamise ja värskendamise protsessi. Täiendavalt on veel kasutusele võetud Fetch API'd, mis kujutab endas brauserisse sisse ehitatud Javascript'i liidest andmete pärimiseks ning vastu võtmiseks [135]. Fetch API erineb tavapärastest AJAX[1] päringutest selle tõttu, et päringuid ei lükata tagasi negatiivse tagastuskoodi kätte saamisel ning ei vahenda küpsiseid [136].

Tagarakenduse API arendusest on juba teada, et eesrakendus peab tegema mitmeid erinevaid päringuid. Selle tõttu on mõistlik luua abiklass, milles on ära kirjeldatud vajalik loogika JSON päringute tegemiseks. Erinevad päringud on jaotatud loogilisteks meetoditeks, mida on võimalik inimloetaval kujul Javascript koodis välja lugeda. Abistava meetodi *useFetch.ts* koodinäide (Koodinäite X) on esitatud Lisas 15, milles API päringu teeb *useFetch()* meetod API-teekonnale GET */api/v1/documents*.

4.2.5 Turvalisus eesrakenduses

Eesrakenduse autentimise võimaldamiseks on kasutusele võetud JWT. Autentimispäringu saatmisel peab eesrakendus JWT'd talletama ning seda järgnevate päringutega kaasa lisama.

---

[1] *Asynchronous Javascript and XML* – tehnoloogiate kogum, mis võimaldab asünkroonseid päringuid [146].

Salvestamaks JWT token'eid (*Access token* ja *Refresh token*) on eesrakenduses kasutusele võetud brauserisse sisse ehitatud *window.localStorage* omadus, mis lubab ligipääsu dokumendi *Storage* objektile (Koodinäide 5). JWT konfiguratsioon ja kehtivuste eest hoolitseb tagarakenduse. Eesrakendus peab päringutega kaasa panema JWT token'i (*Access token*) ning päringute autentimise ebaõnnestumisel suunatakse kasutaja JWT token'it uuendama läbi uuendamise token'i (*Refresh token*). Kui JWT token'i uuendamine ebaõnnestub, siis suunatakse kasutaja sisse logimise lehele. Autoriseerimise läbikukkumise puhul kuvatakse kliendile ekraanil inimloetav veatekst

```
const saveUserAuthDetails = (value: IJwtTokens) => {
  localStorage.setItem(EStorageKey.AUTH, JSON.stringify(value));
};

const getUserAuthDetails = (): IJwtTokens => {
  const value: any = localStorage.getItem(EStorageKey.AUTH);
  return JSON.parse(value);
};
```

Koodinäide 5. JWT talletamine klientrakenduses

4.3 Testimine

Käesolevas lõputöös ei tähenda testimine mitte manuaalset testimist, vaid automaatteste, mis on kirjatud tagarakenduse jaoks. Automaatteste peetakse tarkvaraarenduse oluliseks osaks, sest sellega kaasnevad mitmed head eelised. Automatiseerimine muudab üksikute testide või testkomplektide kiire ja hõlpsa kordamise arenduse ajal lihtsaks ning annab arendajale kohest tagasisidet. See aitab tagada, et väljalasked vastavad kvaliteedi- ja toimivusnõuetele ja veenduda, et kood täidab oma otstarvet. Automatiseerimine suurendab nii üksikute arendajate tootlikkust kui ka tagab, et teste käitatakse arendustegevuse elutsükli kriitilistes punktides, nagu lähtekoodi kontrolli registreerimine, funktsioonide integreerimine ja versiooni väljalase.

Tagarakenduses on automaattestide osakaal suur ning kõik kontrolleri meetodid on kaetud integratsiooni testidega ja nende poolt sisemiselt käivitatud teenuste meetodid on kaetud ühikutestidega (*unit test*). Järgnevalt on kirjeldatud testide kirjutamiseks kasutatud lähenemist ning tehnoloogiad.

4.3.1 Testipõhine arendamine

Rakenduste arendamisel on kasutatud testipõhist tarkvaraarenduse lähenemisviisi, ehk TDD[1]. Tarkvaraarenduses hakkas lähenemisviisi levima aastal 2003, kui Kent Beck'i poolt tutvustati uudset tarkvaraarenduse praktikat, mis keskendub funktsionaalsuste testjuhtumite loomisele enne tarkvara koodi täielikku väljatöötamist [136]. See on iteratiivne tarkvara arendusprotsess, mis ühendab programmeerimise, ühikutestide (*unit test*) loomise ja ümbertöötamise. Protsessi jälgitakse pidevalt kogu tarkvaraarenduse vältel, testides tarkvara korduvalt kõigi testjuhtumite suhtes. See on vastupidine esmalt arendatavale tarkvarale ja hiljem loodud testjuhtumitele. Lihtsamalt öeldes luuakse ja testitakse esmalt iga funktsionaalsuse testjuhtumid ning kui test ebaõnnestub, siis kirjutatakse uus kood, et testi läbida ja kood oleks lihtne ja veavaba.

4.3.2 Tagarakenduse automaattestid

NestJS raamistikuga tuleb kaasa automaattestide raamistik, mis võimaldab mugavalt luual lõppkasutusteste (*E2E*[2]), ühikuteste (*unit test*) ning integratsiooni teste. NestJS on vaikimisi integreeritud levinud ning võimeka Javascript testraamistikuga Jest, mis on hästi dokumenteeritud, lihtne konfigureerida ja seda saab vajadustele vastavaks laiendada. Jest võimaldab kasutada põhilisi testimisfunktsionaalsusi nagu *mocking*[3], *stubbing*[4] ja *spying*[5], mis muudavad testide kirjutamise kui ka lugemise väga lihtsaks.

Automaattestide sooritamisel tehakse reaalseid andmebaasi päringuid, mis võimaldavad täpselt imiteerida funktsioonide käitumist koos andmebaasi kihiga. Automaattestide puhul on andmebaasi kihina võetud kasutusele *In-memory*[6] andmebaas, mis võimaldab jooksvalt ehitada isoleerituid andmebaase ning tagada kiiremat töötlusaega võrreldes kõvaketal olevate andmebaasidega. [128] Iga automaattesti iteratsiooni ajal ehitatakse uus andmebaas olemi-andmeobjektide põhjal ning sooritakse vajalikud andmetöötlused. Peale testi lõpetamisest hävitatakse andmebaas mälust. Lisas 16 on leitav Jest'i põhised

---

[1] *Test Drive Development* – testipõhine tarkvaraarenduses kasutatav lähenemisviis

[2] *End-to-End tests*

[3] Automaattestimise termin, mis tähendab reaalsete objektide ning nende käitumise imiteerimist

[4] Automaattestimise termin, mis tähendab reaalsete objektide imiteerimist

[5] Automaattestimise termin, mis tähendab reaalsete funktsioonide jälgimist

[6] Mälusisesed andmebaasid, mis tuginevad andmete salvestamiseks peamiselt mälule

automaattestide koodinäite (Koodinäide 13), mis kontrollib erandite tagastamist dokumentide detailandmete pärimisel ning dokumenti detailandmete tagastusväärtust.

4.3.3 Eesrakenduse automaattestid

Kliendipoolsed automaattestid on React rakenduse spetsiifilised. Levinud on kasutada React testimiseks Jest testimise raamistikku, mis hoolitseb koodi jooksutamise ning testide valideerimise eest. Käesoleva lõputöö skoobis ei keskenduda kliendipoolse eesrakenduse loomisele ning sel põhjusel ei looda ka vastavaid automaatteste.

4.4 Pidev integratsioon

Pideva integratsioon ehk teisisõnu *Continuous Integration* (CI) on praktika, kus ühe koodibaasiga töötavad arendajad teevad pidevalt koodibaasi muudatusi. CI põhiline eesmärk on võimalikult hõlpsalt jagada kliendibaasile uuendatud rakendust. Selle jaoks kasutatakse tihtipeale automaatteste, et veenduda rakenduse põhifunktsionaalsustes, aga ka automaatseid rakenduse ehitamisvahendeid, mis võivad ehitada ja üles laadida uue keskkonna. Populaarsemad tööriistad on Bamboo, Jenkins ja CircleCI.

4.4.1 Bitbucket Pipeline

Käesolevas arendusprotsessis on kasutatud Bitbucket'i pakutud CI tööriista, mis võimaldab nii automaattestide jooksutamist kui ka automaatset rakenduse paigaldust testserverisse. Selleks pakub Bitbucket *pipeline'i* ühendust, kus saab täpsustada oma testserverit, reaalset serverit ning luua ühenduse Bitbucket'i repositoorimiga. Muudatuste lisamisel põhikoodibaasi ehitatakse teenusele uus Docker'i tõmmisfail (*image*), mis seejärel laetakse ülese avaliku tõmmiste registrisse Docker Hub ning lõpuks paigaldatakse ja taaskäivitakse kogu tagarakendus testserveris.

4.4.2 Tagarakenduse serveerimine

Projektis on kasutatud tagarakenduse testserveri majutamiseks ja ning domeenikirjete haldamiseks DigitalOcean'i teenuspakkujat. Domeeninimi *tarviraun.eu* on registreeritud Eesti DNS[1] teenuspakkuja Zone.eu juures ning domeenikirjete mugavaks haldamiseks on

[1] Domain Name System – detsentraliseeritud nimetussüsteem seadetele, mis on ühendatud internetiga

suunatud nimeserver DigitalOcean'sse. Tagarakenduse kättesaadavaks tegemiseks veebiaadressil *api.tarviraun.eu/api* on seadistatud domeeni IP-põhine A-kirje ning domeeninimede vahelise liikluse seadistamiseks on määratud vastav CNAME-kirje[1]. Tagarakendus on käesolevas skoobis seadistatud testserveris *IaaS* pilvelahendusena.

[1] Canonical Name Record – DNS'i ressurss, mis täpsustab ühte domeeni teise aliasena.

5 Hinnang loodud Restful-API rakendusele

Käesolevas peatükis antakse ülevaade lõputöö praktilises osas valminud mikroteenustena arendatud Restful-API rakendusele. Loodud API-rakendusliidesest saab kõige paremini hinnata selle funktsionaalsuste alusel. Samuti saab hinnangut anda analüüsides testide kattuvust, täidetud nõudeid, edasiarendatavust ning kasutatud tehnoloogiate jätkusuutlikkust.

5.1 Saavutatud tulemused

Saadud tulemuste valideerimiseks hinnati loodud rakenduse vastavust analüüsi käigus püstitatud funktsionaalsetele ja mittefunktsionaalsetele nõuetele. Kõik mittefunktsionaalsed ning enamus funktsionaalseid nõudeid said projekti arendamise jooksul valmis. Lahenduse arendamisel oli väga oluline osa rakenduse liidestamisel Dokobit Gateway API'ga ning Confluence Cloud API'ga. Väljakutseks osutus dokumenti failide töötlemine ning nende edastamine erinevate süsteemide vahel. Teine väljakutse oli rakenduse kasutajakontode loomine ja haldamine ilma kasutajapoolse registreerimiseta. Käesolevas rakenduses luuakse ning tuvastatakse kasutajakontod automaatselt kasutades selleks Atlassian ID kasutajakontot.

Lõputöö skoop ei sisaldanud kliendipoolse eesrakenduse loomist, mis küll kuulub lõputöö konteksti aga on planeeritud järgmisse arendusetappi. Tulevikus loodava Forge'i kliendipoolse eesrakenduse vajaduste ja nõuete väljaselgitamiseks installeeriti Confluence Cloud tarkvaraplatvormil Forge rakendus ning juurutati osaliselt ka API tagarakendus tulevikus kasutavate lisandmoodulitega. Kliendipoolse eesrakenduse osalisel liidestamisel oli esmaseks eesmärgiks arvestada arendamisel varakult REST API kutsete ja vastuste disainimisel Forge eripärade ja vajadustega. Teine eesmärk oli valideerida ja täpsustada analüüsi etapis kirjeldatud kasutajaliidesega seotud nõuete ja kasutajalugude realiseeritavus.

Lõputöös kujunes ajamahukaks disainitud mikroteenuste arhitektuurile vastava rakenduse loomine koos gRPC transpordikihiga. Arendavale lahendusele sai püstitud tuleviku perspektiiviga nõudvaid mittefunktsionaalseid nõudeid ning seetõttu on loodud Restful API rakendus olemuselt keerukas. Täidetud said kõik püstitatud nõuded ja eeldused skaleeritavuse, paindlikkuse ja kohaldamise osas. Samuti on võimalik loodud

rakendust mugavalt laiendada ning luua liideseid teiste dokumendihaldussüsteemide ning allkirjastamise teenustega integreerimiseks.

5.2 Koodi valideerimine

Rakenduse töökindluse tagamiseks ning jätkuarendusel kvaliteedi pidevaks hoidmiseks on suur osa koodist kaetud ühik– ning integratsioonitestidega. Automaattestid on samuti hea viis koodi dokumenteerimiseks, kuna need kirjeldavad hästi, mida testitav funktsionaalsus koodis peaks tegema ning milline on ootuspärane käitumine. Rakenduste arendamisel on kasutatud testipõhist tarkvaraarenduse lähenemisviisi, ehk TDD'd.

Lõputöö arendamisel osutus automaattestide kirjutamine ja kirjeldamine ajakulukaks tegevuseks. Keerukaks osutus gRPC protseduurkutse klient-serveri ning *in-memory* andmebaasi ehitamine iga integratsioonitesti iteratsiooni jaoks.

5.3 Kasutatud tehnoloogiate uudsus

Lõputöö lahenduse arendamisel sai teadlikult tehtud otsus, et kasutatakse vaid laialt kasutatud standardeid ning tehnoloogiaid. Kaasaegsed ning populaarsed tehnoloogiad, mida kasutati käesoleva lõputöös, arenevad pidevalt edasi ning arendajana on oluline hoida ennast kursis kõige värskemate trendidega.

5.4 Võimalikud edasiarendused

Järgmise arendusetapi skoobis on tähtsaim prioriteed kliendipoolse kasutajaliidese loomine Forge eesrakendusena ning implementeerimine loodud Restful API'ga. Kliendipoolsed funktsionaalsused on analüüsi etapis kirjeldatud kasutajalugudena. Kliendipoolse kasutajaliidese arenduse skoop on planeeritud realiseerida kahe erineva etapina. Esimeses etapis arendatakse funktsionaalsused, mis on seotud allkirjastavate dokumentide loomisega, allkirjastamisega ning haldamisega. Teises etapis arendatakse juurde uued vaated administreerimistööriistade tarvis, mis võimaldavad hallata kasutajaid ning neile määratud tegevuste õigusi rollide kaudu.

Äriliselt on kasumlik teenuste portfelli laiendada ning luua integratsioone ka teiste dokumendihaldussüsteemidega lisaks juba loodud liidesele Confluence Cloud'iga. Järgmised dokumendihaldussüsteemid, millele luua liidesed integreerimiseks, oleksid

esialgses järjekorras Google Drive, Dropbox ning Microsoft SharePoint. Täpsema tasuvuse kaardistamiseks ja arendamise prioriteetide välja selgitamiseks on vajalik läbi viia täiendav analüüs.

Tulevikku jätkutegevustes on planeeritud mitmeid DevOps'ga seotud tehnoloogilisi uuendusi. Suureneva koormusega toimetulekuks on plaanis viia lõpuni varem kirjeldatud tagarakenduse halduse üleminek *PaaS* teenusekesksele arhitektuurile. Skaleeritavuse tagamiseks on plaanitud pilveorkestraatorina1 kasutusele võtta Kubernetes'e tehnoloogia, mis võimaldab automaatset teenuste koordineerimist ja haldamist [130]. Infrastruktuurilise konfiguratsiooni haldamiseks ning teenuspakkujale varustamiseks on tulevikus plaanis kasutada vabavaralist IaC[2] tööriista Terraform'i, mis võimaldab koodina deklaratiivselt kirjeldada vajalikke seadistusi teenuste automaatseks haldamiseks [131].

[1] Pilveorkestraator (*cloud orchestrator*) - tarkvara, mis haldab ühendusi ja interaktsioone pilvepõhiste ja kohalike talitlusüksuste, protsesside ja töövoogude vahel.

[2] IaC (*Infrastructure as Code*) – kõrgetasemeline kodeerimiskeelt, mis võimaldab IT-infrastruktuuri konfiguratsiooni varustada deklaratiivse koodina [148].

6 Kokkuvõtte

Lõputöö eesmärgiks oli luua lahendus, mis aitaks rakendata digiallkirjastamist Eestis kasutavates meditsiiniseadmete tootjate kvaliteedijuhtimissüsteemides. Analüüsi käigus uuriti põhjalikult meditsiiniseadmete tootmisvaldkonda puututavaid regulatsioone nii kvaliteedijuhtimissüsteemide kui allkirjastamise kohta. Analüüsi tulemusel otsustati lahendus ehitada tarkvaraplatvormil Confluence Cloud'l lisades sellele vajalikud funktsionaalsused, mis võimaldaks rakendada Eestis juriidiliselt kehtivaid digitaalselt allkirjastatud dokumente. Käesoleva lõputöö skoobiks oli seega Restful API tagarakenduse loomine, mis pakub teenuseid tulevikus arendavale Confluence Cloud eesrakendusele. Töö käigus loodi ka vajalikud eeldused järgmises etapis arendatavale kasutajaliidesele.

Käesolevas lõputöö ülesande püstitusest andis hea ülevaate analüüsi peatükk, mis esmalt kaardistas ära lähtetingimused tehtavale tööle. Planeeritavad funktsionaalsused selgitati välja MoSCoW meetodi läbi funktsionaalsete nõuete prioritiseerimise abil ning mittefunktsionaalseid nõudeid kirjeldati FURPS+ mudeli kaudu. Samuti kirjeldati kasutuslugude kaudu ära tulevikus loodava eesrakenduse funktsionaalsused. Analüüsi peatüki teises osas kirjeldati detailselt eesmärgi saavutamiseks kasutatavaid tehnoloogiad ning põhjendati vastavate valikute tegemist. Lõputöö arenduskäiku kirjeldav peatükk andis ülevaate nii tagarakenduse arhitektuurist, teenuste kihtidest, turvalisusest ning loodavast lahendustest.

Lõputöö projekti tulemusena valmis mikroteenuste arhitektuuril põhinev Restful API tagarakendus, mis vastab kõigile tagarakendusele püstitatud nõuetele. Tagarakendus ehitati raamistikule NestJS, mis kirjutati TypeScript'i programmeerimiskeeles. Loodud tagarakendus järgib REST API põhimõtteid ning on dokumenteeritud OpenAPI standardi alusel. Järgmisena arendatava eesrakenduse sujuvamaks jätkuarenduseks loodi kasutajaliidese funktsionaalsusi kirjeldavad kasutajalood, analüüsiti kasutatavaid tehnoloogiad ning seadistati eesrakendus mittefunktsionaalsete nõuete alusel.

Tagarakenduse loomisel arvestati tuleviku perspektiiviga ning edasiarendamise ja laiendamise lihtsustamiseks oli mikroteenustel põhinev arhitektuur eelnevalt põhjalikult läbimõeldud ning saavutamiseks püstitati vajalikud mittefunktsionaalsed nõuded. Suur osa koodist kaeti ühik– ning integratsioonitestidega, mis aitavad tagada jätkuarendustel

rakenduse töökindlust ja kvaliteeti ning on heaks viisiks olemasoleva koodi dokumenteerimiseks ja ootuspärase käitumise kirjeldamiseks.

Projekti võib lugeda õnnestunuks, kuna tagarakenduses realiseeriti analüüsis püstitatud nõuded ning tagarakendust on võimalik liidestada tarkvaraplatvormiga Confluence Cloud. Samuti on lõpuni viidud vajalikud eeltööd järgmise arendusetapis realiseeritava eesrakenduse loomiseks. Kuigi eesrakenduse funktsionaalsused ei valminud arendusprotsessi käigus tänu töö mahukusele, on pandud tugev alus tagarakenduse edasisele arendusele ning eesrakenduse loomisele. Antud lõputöö raames õppis autor palju uusi tehnoloogiaid ja sai eelnevalt omandatud teadmisi proovile panna. Pikemas perspektiivis oli eesmärk omandada kogemusi reguleeritud valdkondades tarkvaratoodete loomiseks ja arendamiseks.

Kasutatud kirjandus

[1] EUR-Lex Publications Office of the European Union, „EUROOPA PARLAMENDI JA NÕUKOGU MÄÄRUS (EL) 2017/745,“ 24 märts 2020. [Võrgumaterjal]. Kättesaadav: https://eur-lex.europa.eu/legal-content/ET/TXT/HTML/?uri=CELEX:02017R0745-20200424&from=EN.

[2] International Organization for Standardization, „ISO 13485 Medical devices,“ [Võrgumaterjal]. Kättesaadav: https://www.iso.org/iso-13485-medical-devices.html. [Kasutatud 10 03 2023].

[3] Parasoft, „What Is IEC 62304 & How Is It Used in Medical Device Compliance?,“ 20 05 2021. [Võrgumaterjal]. Kättesaadav: https://www.parasoft.com/blog/what-is-iec-62304-how-is-it-used-in-medical-device-compliance/.

[4] A. Velling, „5 ISO standardit tööstuses,“ 12 10 2018. [Võrgumaterjal]. Kättesaadav: https://fractory.com/et/5-iso-standardit-toostuses/.

[5] George Washington University, „The Difference between Electronic and Paper Documents,“ [Võrgumaterjal]. Kättesaadav: https://www2.seas.gwu.edu/~shmuel/WORK/Differences/The%20Difference%20between%20Electronic%20and%20Paper%20Documents.html. [Kasutatud 10 märts 2023].

[6] EUR-Lex, „E-identimise ja e-tehingute usaldusteenuste määrus,“ [Võrgumaterjal]. Kättesaadav: https://eur-lex.europa.eu/legal-content/ET/TXT/HTML/?uri=CELEX%3A32014R0910&from=EN. [Kasutatud 12 märts].

[7] EUR-Lex, „E-identimise ja e-tehingute usaldusteenuste määrus,“ [Võrgumaterjal]. Kättesaadav: https://eur-lex.europa.eu/legal-content/ET/TXT/HTML/?uri=CELEX%3A32014R0910&from=EN. [Kasutatud 12 03].

[8] Riigi Infosüsteemi Amet, „Digitaalne allkirjastamine ja elektroonilised allkirjad,“ [Võrgumaterjal]. Kättesaadav: https://www.id.ee/artikkel/digitaalne-allkirjastamine-ja-elektroonilised-allkirjad/. [Kasutatud 12 märts 2023].

[9] Riigi Infosüsteemi Amet, „DigiDoc4 klient,“ [Võrgumaterjal]. Kättesaadav: https://www.id.ee/rubriik/digidoc4-klient/. [Kasutatud 04 aprill 2023].

[10] Dokobit, „Dokumentide allkirjastamine,“ [Võrgumaterjal]. Kättesaadav: https://www.dokobit.com/et/. [Kasutatud 02 aprill 2023].

[11] Dokobit, „Dokumentide API,“ [Võrgumaterjal]. Kättesaadav: https://www.dokobit.com/et/lahendused/dokumentide-api. [Kasutatud 02 aprill 2023].

[12] Adobe, „Adobe Acrobat Sign FAQ,“ [Võrgumaterjal]. Kättesaadav: https://helpx.adobe.com/sign/faq.html. [Kasutatud 02 aprill 2023].

[13] Adobe, „Electronic Signature Laws & Regulations - The European Union,“ [Võrgumaterjal]. Kättesaadav: https://helpx.adobe.com/legal/esignatures/regulations/european-union.html. [Kasutatud 02 aprill 2023].

[14] Andmekaitse Inspektsioon, „Isikuandmete liigitus,“ [Võrgumaterjal]. Kättesaadav: https://www.aki.ee/et/eraelu-kaitse/isikuandmed-ja-tootlemine/isikuandmete-liigitus. [Kasutatud 03 aprill 2023].

[15] Andmekaitse Inspektsioon, „Vastutav ja volitatud töötleja,“ [Võrgumaterjal]. Kättesaadav: https://www.aki.ee/et/vastutav-ja-volitatud-tootleja. [Kasutatud 03 aprill 2023].

[16] M. Nadeau, „General Data Protection Regulation (GDPR): What you need to know to stay compliant,“ 12 juuni 2020. [Võrgumaterjal]. Kättesaadav: https://www.csoonline.com/article/3202771/general-data-protection-regulation-gdpr-requirements-deadlines-and-facts.html.

[17] T. Mõttu, „Viis sammu andmekaitsemääruse täitmiseks,“ 13 juuni 2017. [Võrgumaterjal]. Kättesaadav: https://www.primend.ee/blogi/viis-sammu-andmekaitsemaaruse-taitmiseks/.

[18] Euroopa Komisjon, „Kes on vastutav töötleja või volitatud töötleja?,“ 17 mai 2018. [Võrgumaterjal]. Kättesaadav: https://commission.europa.eu/law/law-topic/data-protection/reform/rules-business-and-organisations/obligations/controllerprocessor/what-data-controller-or-data-processor_et.

[19] OneTrust, „HIPAA vs. GDPR compliance: what’s the difference?,“ 21 september 2022. [Võrgumaterjal]. Kättesaadav: https://www.onetrust.com/blog/hipaa-vs-gdpr-compliance.

[20] Atlassian, „Marktetplace - Pricing, payment, and billing,“ [Võrgumaterjal]. Kättesaadav: https://developer.atlassian.com/platform/marketplace/pricing-payment-and-billing/. [Kasutatud 01 aprill 2023].

[21] R. B. Dai Clegg, Case Method Fast-Track: A Rad Approach, Boston: Addison-Wesley Publishing Company, 1994.

[22] Agile Business Consortium Limited, „MoSCoW Prioritisation,“ [Võrgumaterjal]. Kättesaadav: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html. [Kasutatud 10 aprill 2023].

[23] M.-C. Lee, Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance, British Journal of Applied Science and Technology, 2014.

[24] M. S. S. T. A. Al-Badareen, A Comparative Study of Software Quality Models, Kuantan: Software Engineering and Computer Systems: Second International Conference, 2011.

[25] IBM, „Nonfunctional requirements: A checklist,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/cloud/architecture/architecture/practices/nonfunctional-requirements-checklist/. [Kasutatud 18 märts 2023].

[26] RestCase, „The Rise of REST API,“ 20 oktoober 2015. [Võrgumaterjal]. Kättesaadav: https://blog.restcase.com/the-rise-of-rest-api/.

[27] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,“ Dissertation , 2000.

[28] DevelopMentor, „Simple Object Access Protocol (SOAP) 1.1,“ 08 mai 2000. [Võrgumaterjal]. Kättesaadav: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[29] Majandus-ja Kommunikatsiooniministeerium, „Riigi IT arhitektuur,“ 2007. [Võrgumaterjal]. Kättesaadav: https://web.archive.org/web/20210111191541/https://www.mkm.ee/sites/default/files/riigi_it_arhitektuur.pdf.

[30] Stack Overflow, „2021 Developer Survey,“ 2021. [Võrgumaterjal]. Kättesaadav: https://insights.stackoverflow.com/survey/2021.

[31] R. Toal, „What is PHP?,“ [Võrgumaterjal]. Kättesaadav: https://codeinstitute.net/global/blog/what-is-php-programming/. [Kasutatud 7 aprill 2023].

[32] IBM, „What is Java?,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/topics/java. [Kasutatud 7 aprill 2023].

[33] Microsoft, „Microsoft docs,“ [Võrgumaterjal]. Kättesaadav: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp. [Kasutatud 7 aprill 2023].

[34] Python Software Foundation, „What is Python? Executive Summary,“ [Võrgumaterjal]. Kättesaadav: https://www.python.org/doc/essays/blurb/. [Kasutatud 7 aprill 2023].

[35] OpenJS Foundation, „About Node.js,“ [Võrgumaterjal]. Kättesaadav: https://nodejs.dev/learn/introduction-tonodejs/. [Kasutatud 7 aprill 2023].

[36] GeeksForGeeks, „How to Choose a Programming Language For a Project?,“ [Võrgumaterjal]. Kättesaadav: https://www.geeksforgeeks.org/how-to-choose-a-programming-language-for-a-project/. [Kasutatud 7 aprill 2023].

[37] LogRocket, „Comparing top Node.js frameworks for frontend developers,“ 2022. [Võrgumaterjal]. Kättesaadav: https://blog.logrocket.com/comparing-top-node-js-frameworks-frontend-developers/. [Kasutatud 7 aprill 2023].

[38] K. Kreuzer, „Angular & Nest, a match made in heaven,“ 2020. [Võrgumaterjal]. Kättesaadav: https://kevinkreuzer.medium.com/angular-nest-a-match-made-in-heaven-e52cb8e4105a/.

[39] A. Rump, „Versions of .NET Frameworks,“ 2023. [Võrgumaterjal]. Kättesaadav: https://versionsof.net/framework.

[40] A. Rump, „Versions of .NET Core,“ 2023. [Võrgumaterjal]. Kättesaadav: https://versionsof.net/core.

[41] InterviewBit, „.NET Core vs .NET Framework,“ 29 11 2022. [Võrgumaterjal]. Kättesaadav: https://www.interviewbit.com/blog/net-core-vs-net-framework.

[42] Stack Overflow, „2022 Developer Survey,“ 2022. [Võrgumaterjal]. Kättesaadav: https://survey.stackoverflow.co/2022/#technology.

[43] InterviewBit, „Top Java Frameworks You Must Know in 2023,“ 2023. [Võrgumaterjal]. Kättesaadav: https://www.interviewbit.com/blog/java-frameworks. [Kasutatud 7 aprill 2023].

[44] N. Telsan, „Node Package Managers in 2022,“ 2022. [Võrgumaterjal]. Kättesaadav: https://www.viget.com/articles/node-package-managers-in-2022/. [Kasutatud 7 aprill 2023].

[45] Cabot Technology Solution, „Hackernoon - How to Pick the Right Web Technology Stack for Your Product.,“ märts 2017. [Võrgumaterjal]. Kättesaadav: https://hackernoon.com/how- to-pick-the-right-web-technology-stack-for-your-product-f6d94440af2f.

[46] W. Dunkley, „Medium - Split Stack Development: A Model For Modern Applications.,“ juuni 2016. [Võrgumaterjal]. Kättesaadav: https://medium.com/@MentallyFriendly/split- stack-development-a-model-for-modern-applications-d7b9abb47bd5.

[47] Forbes Technology Council, „Seven Reasons Why A Website's Front-End And Back-End Should Be Kept Separate,“ 19 juuli 2018. [Võrgumaterjal]. Kättesaadav: https://www.forbes.com/sites/forbestechcouncil/2018/07/19/seven-reasons-why-a-websites-front-end-and-back-end-should-be-kept-separate/?sh=584f6edd4fca.

[48] A. Ariscrisnã, „Choosing the best tech stack for web development 2023,“ 20 märts 2023. [Võrgumaterjal]. Kättesaadav: https://www.imaginarycloud.com/blog/choosing-the-best-tech-stack-for-web-development-2022/.

[49] Object Computing Inc. , „Micronaut,“ [Võrgumaterjal]. Kättesaadav: http://micronaut.io/. [Kasutatud 9 aprill 2023].

[50] Kinsta, „What Is Nest.js? A Look at the Lightweight JavaScript Framework,“ 21 november 2022. [Võrgumaterjal]. Kättesaadav: https://kinsta.com/knowledgebase/nestjs/.

[51] C. Balola, „Building a secure and quality NodeJS REST API with NestJS,“ 18 07 2022. [Võrgumaterjal]. Kättesaadav: https://medium.com/@chrisbalola/building-a-secure-and-quality-nodejs-rest-api-with-nestjs-34a9aef633e0.

[52] O. Romanyuk, „Express.js Security Tips: How You Can Save and Secure Your App,“ 10 10 2019. [Võrgumaterjal]. Kättesaadav: https://www.freecodecamp.org/news/express-js-security-tips/.

[53] V. Sharma, „Express Vs Fastify,“ 10 september 2021. [Võrgumaterjal]. Kättesaadav: https://medium.com/@vishalims095/express-vs-fastify-9a4d052c28e1.

[54] B. F. Aginsa, „Fastify vs Express Performance,“ 10 jaanuar 2022. [Võrgumaterjal]. Kättesaadav: https://facsiaginsa.com/nodejs/comparing-fastify-vs-express.

[55] M. Tria, „Introducing Forge, a new way to build and run apps for the Atlassian cloud,“ 12 detsember 2019. [Võrgumaterjal]. Kättesaadav: https://www.atlassian.com/blog/announcements/introducing-forge.

[56] Atlassian, „Build with Forge,“ [Võrgumaterjal]. Kättesaadav: https://developer.atlassian.com/platform/forge/. [Kasutatud 8 aprill 2023].

[57] W3Schools, „What is React?,“ [Võrgumaterjal]. Kättesaadav: https://www.w3schools.com/whatis/whatis_react.asp. [Kasutatud 08 aprill 2023].

[58] Atlassian, „Forge - User Interface,“ [Võrgumaterjal]. Kättesaadav: https://developer.atlassian.com/platform/forge/user-interface/. [Kasutatud 08 aprill 2023].

[59] Atlassian, „Forge Bridge,“ [Võrgumaterjal]. Kättesaadav: https://developer.atlassian.com/platform/forge/custom-ui/#bridge. [Kasutatud 08 aprill 2023].

[60] InfoWorld, „Microsoft augments JavaScript for large-scale development,“ 01 oktoober 2012. [Võrgumaterjal]. Kättesaadav: https://www.infoworld.com/article/2614863/microsoft-augments-javascript-for-large-scale-development.html.

[61] A. Masan, „Understanding and using interfaces in TypeScript,“ 17 aprill 2022. [Võrgumaterjal]. Kättesaadav: https://blog.logrocket.com/understanding-using-interfaces-typescript/.

[62] Orta.io, „Understanding TypeScript's Popularity,“ [Võrgumaterjal]. Kättesaadav: https://orta.io/notes/js/why-typescript. [Kasutatud 20 aprill 2023].

[63] L. Võsandi, „Lauri blog - Andmebaaside võrldus,“ 2016. [Võrgumaterjal]. Kättesaadav: https://lauri.xn--vsandi-pxa.com/rdbms/comparison.html.

[64] E. Eessaar, Andmebaaside projekteerimine, Tallinna Tehnikaülikooli kirjastus, 2018.

[65] S. Yegulalp, „Why you should use SQLite,“ 13 veebruar 2019. [Võrgumaterjal]. Kättesaadav: https://www.sqlite.org/aff_short.html.

[66] M. Drake, „SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems,“ 21 veebruar 2014. [Võrgumaterjal]. Kättesaadav: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems.

[67] Oracle Patches, „MySQL Database Overview - Strengths and Weakness of MySQL,“ 25 märts 2018. [Võrgumaterjal]. Kättesaadav: https://oracle-patches.com/en/databases/mysql/overview-of-mysql-database?start=2.

[68] K. Hristozov, „MySQL vs PostgreSQL -- Choose the Right Database for Your Project,“ 19 juuli 2019. [Võrgumaterjal]. Kättesaadav: https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres.

[69] Webandcrafts, „Advantages and Disadvantages of MongoDB NoSQL Database: An Ultimate Guide,“ 15 oktoober 2021. [Võrgumaterjal]. Kättesaadav: https://data-flair.training/blogs/advantages-of-mongodb/.

[70] Cubet Techno Labs., „Medium - The Essential Difference Between Couchbase & MongoDB,“ jaanuar 2018. [Võrgumaterjal]. Kättesaadav: https://medium.com/@Cubet_Techno_Labs/the-essential-difference-between-couchbase-mongodb-5b4d7d192cff.

[71] InfluxData, „Key-Value Database: How It Works, Key Features, Advantages,“ 2022. [Võrgumaterjal]. Kättesaadav: https://www.influxdata.com/key-value-database/.

[72] Research Computing University of Virginia, „Redis - A Key/Value Store,“ 02 veebruar 2022. [Võrgumaterjal]. Kättesaadav: https://www.rc.virginia.edu/userinfo/howtos/general/redis/.

[73] P. Raspel, „Andmemudelite normaliseerimine,“ 2022. [Võrgumaterjal]. Kättesaadav: https://enos.itcollege.ee/~priit/1.%20Andmebaasid/1.%20Loengumaterjalid/07/7.htm.

[74] The PostgreSQL Global Development Group, „JSON Types,“ [Võrgumaterjal]. Kättesaadav: https://www.postgresql.org/docs/current/datatype-json.html. [Kasutatud 08 aprill 2023].

[75] C. Gehman, „What Is DVCS?,“ 25 juuli 2018. [Võrgumaterjal]. Kättesaadav: https://www.perforce.com/blog/vcs/what-dvcs.

[76] S. Sharma, „Bitbucket vs Github vs Gitlab,“ 26 detsember 2022. [Võrgumaterjal]. Kättesaadav: https://www.geeksforgeeks.org/bitbucket-vs-github-vs-gitlab/.

[77] Incredibuild, „What is GitLab?,“ [Võrgumaterjal]. Kättesaadav: https://www.incredibuild.com/integrations/gitlab. [Kasutatud 10 aprill 2023].

[78] Atlassian, „Connecting Jira Software and Bitbucket,“ [Võrgumaterjal]. Kättesaadav: https://www.atlassian.com/software/jira/bitbucket-integration. [Kasutatud 09 aprill 2023].

[79] IO Mart, „Advantages and disadvantages of dedicated server hosting,“ 21 september 2022. [Võrgumaterjal]. Kättesaadav: https://www.iomart.com/iolearn/advantages-and-disadvantages-of-dedicated-server-hosting/.

[80] IBM, „What are Iaas, Paas and Saas?,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/topics/iaas-paas-saas. [Kasutatud 08 aprill 2023].

[81] Atlassian, „Microservices vs. monolithic architecture,“ [Võrgumaterjal]. Kättesaadav: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. [Kasutatud 08 aprill 2023].

[82] M. Beznos, „Microservices vs monolith: Which architecture is the best choice for your business?,“ 03 jaanuar 2023. [Võrgumaterjal]. Kättesaadav: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/.

[83] IBM, „What are microservices?,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/topics/microservices. [Kasutatud 15 aprill 2023].

[84] IBM Cloud Team, IBM Cloud, „SOA vs. Microservices: What’s the Difference?,“ 14 mai 2021. [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/cloud/blog/soa-vs-microservices.

[85] Keitaro, „Monoliths vs. Microservices,“ 20 mai 2020. [Võrgumaterjal]. Kättesaadav: https://keitaroinc.medium.com/saas-starter-pack-part-2-monoliths-vs-microservices-58fc51cc228d.

[86] M. Fowler, „Microservice Trade-Offs,“ 01 juuli 2015. [Võrgumaterjal]. Kättesaadav: https://martinfowler.com/articles/microservice-trade-offs.html.

[87] DevelopMentor, „Simple Object Access Protocol (SOAP) 1.1,“ 08 mai 2000. [Võrgumaterjal]. Kättesaadav: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[88] B. Wootton, „What Are Legacy Systems and What Should You Do with Them?,“ 02 oktoober 2017. [Võrgumaterjal]. Kättesaadav: https://www.contino.io/insights/what-are-legacy-systems-and-what-should-you-do-with-them.

[89] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,“ Dissertation, 2000.

[90] Stackify, „SOAP vs. REST: The Differences and Benefits Between the Two Widely-Used Web Service Communication Protocols.,“ 14 mai 2017. [Võrgumaterjal]. Kättesaadav: https://stackify.com/soap-vs-rest/.

[91] L. Gupta, „REST Architectural Constraints,“ 09 märts 2022. [Võrgumaterjal]. Kättesaadav: https://restfulapi.net/rest-architectural-constraints/.

[92] E. Huang, „RESTful API vs Microservice,“ 05 september 2016. [Võrgumaterjal]. Kättesaadav: https://medium.com/@ericjwhuang/restful-api-vs-microservice-eea903ac3e73.

[93] M. Lobjakas, „7 tegurit, mis mõjutavad kasutajakogemust,“ 19 detsember 2017. [Võrgumaterjal]. Kättesaadav: https://blog.twn.ee/et/7-tegurit-mis-mojutavad-kasutajakogemust.

[94] Atlassina, „Common UI kit components,“ [Võrgumaterjal]. Kättesaadav: https://developer.atlassian.com/platform/forge/ui-kit-components/. [Kasutatud 11 aprill 2023].

[95] J. Nielsen, „10 Usability Heuristics for User Interface Design,“ 15 november 2020. [Võrgumaterjal]. Kättesaadav: https://www.nngroup.com/articles/ten-usability-heuristics/.

[96] Atlassian, „Atlassian Design System,“ [Võrgumaterjal]. Kättesaadav: https://atlassian.design. [Kasutatud 09 aprill 2023].

[97] G. Wright, „What is agnostic?,“ [Võrgumaterjal]. Kättesaadav: https://www.techtarget.com/whatis/definition/agnostic. [Kasutatud 15 aprill 2023].

[98] M. Fowler, „Microservices.html,“ 25 märts 2014. [Võrgumaterjal]. Kättesaadav: https://martinfowler.com/articles/microservices.html.

[99] IBM, „What is Docker?,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/topics/docker. [Kasutatud 19 aprill 2023].

[100] S. Novak, „Advantages of Docker for Microservices: Detailed Guide,“ 29 märts 2023. [Võrgumaterjal]. Kättesaadav: https://intellisoft.io/docker-and-microservices-the-future-of-scalable-and-resilient-application-development/.

[101] Microservices.io, „Pattern: API Gateway / Backends for Frontends,“ [Võrgumaterjal]. Kättesaadav: https://microservices.io/patterns/apigateway.html. [Kasutatud 19 aprill 2023].

[102] gRPC Authors, „What is gRPC?,“ [Võrgumaterjal]. Kättesaadav: https://grpc.io/docs/what-is-grpc. [Kasutatud 19 aprill 2023].

[103] J. Hillpot, „gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs?,“ 11 november 2022. [Võrgumaterjal]. Kättesaadav: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/.

[104] Google LLC, „Protocol Buffers Documentation,“ 2023. [Võrgumaterjal]. Kättesaadav: https://protobuf.dev/overview.

[105] J. H, „gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs?,“ 11 11 2022. [Võrgumaterjal]. Kättesaadav: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/.

[106] IBM, „What is three-tier architecture?,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/topics/three-tier-architecture. [Kasutatud 15 aprill 2023].

[107] Sboro, „Nest.js — Architectural Pattern, Controllers, Providers, and Modules.,“ 15 mai 2021. [Võrgumaterjal]. Kättesaadav: https://medium.com/geekculture/nest-js-architectural-pattern-controllers-providers-and-modules-406d9b192a3a.

[108] NestJS, „CLI command reference,“ [Võrgumaterjal]. Kättesaadav: https://docs.nestjs.com/cli/usages. [Kasutatud 15 04 2023].

[109] Bamboo Agile, „What Is the Best Database for Node.js?,“ [Võrgumaterjal]. Kättesaadav: https://bambooagile.eu/insights/the-best-database-for-node-js/. [Kasutatud 16 aprill 2023].

[110] NestJS Docs, „NestJS Databases,“ [Võrgumaterjal]. Kättesaadav: https://docs.nestjs.com/techniques/database. [Kasutatud 16 04 2023].

[111] TypeORM docs, „TypeORM Migrations,“ [Võrgumaterjal]. Kättesaadav: https://orkhan.gitbook.io/typeorm/docs/migrations. [Kasutatud 16 aprill 2023].

[112] Knex.js guide, „Knex.js - Migrations,“ [Võrgumaterjal]. Kättesaadav: https://knexjs.org/guide/migrations.html. [Kasutatud 16 aprill 2023].

[113] C. Custer, „What is a UUID, and what is it used for?,“ 16 märts 2023. [Võrgumaterjal]. Kättesaadav: https://www.cockroachlabs.com/blog/what-is-a-uuid/.

[114] The PostgreSQL Global Development Group, „uuid-ossp Functions,“ [Võrgumaterjal]. Kättesaadav: https://www.postgresql.org/docs/current/uuid-ossp.html. [Kasutatud 15 aprill 2023].

[115] P. S. S. R. Martin Schaffe, „Universally Unique Identifiers: How to ensure Uniqueness while protecting the Issuer’s Privacy,“ 2017. [Võrgumaterjal]. Kättesaadav: https://www.syssec.at/user/themes/syssec-theme/pdfs/UUID_2007.pdf.

[116] RESTfulAPI.net, „REST API Tutorial - REST Resource Naming Guide,“ 20 detsember 2022. [Võrgumaterjal]. Kättesaadav: https://restfulapi.net/resource-naming/.

[117] L. Gupta, „HTTP Status Codes,“ 17 detsember 2021. [Võrgumaterjal]. Kättesaadav: https://restfulapi.net/http-status-codes/.

[118] Noname Security, „What is OpenAPI?,“ [Võrgumaterjal]. Kättesaadav: https://nonamesecurity.com/learn-what-is-openapi. [Kasutatud 16 aprill 2023].

[119] SmartBear, „What Is Swagger?,“ [Võrgumaterjal]. Kättesaadav: https://swagger.io/docs/specification/2-0/what-is-swagger/. [Kasutatud 16 aprill 2023].

[120] Dokobit, „Dokumentide API,“ [Võrgumaterjal]. Kättesaadav: https://www.dokobit.com/et/lahendused/dokumentide-api. [Kasutatud 02 aprill 2023].

[121] A. Magnusson, „What is Authentication?,“ 13 veebruar 2023. [Võrgumaterjal]. Kättesaadav: https://www.strongdm.com/authentication.

[122] Internet Engineering Task Force (IETF), „JSON Web Token (JWT),“ 2015. [Võrgumaterjal]. Kättesaadav: https://datatracker.ietf.org/doc/html/rfc7519.

[123] Kinsta, „Environment Variables: What They Are and How To Use Them,“ 08 märts 2023. [Võrgumaterjal]. Kättesaadav: https://kinsta.com/knowledgebase/what-is-an-environment-variable/.

[124] NestJS, „NestJS - Validation,“ [Võrgumaterjal]. Kättesaadav: https://docs.nestjs.com/techniques/validation. [Kasutatud 19 aprill 2023].

[125] Tanner Linsley, „TanStack Query overview,“ [Võrgumaterjal]. Kättesaadav: https://tanstack.com/query/latest/docs/react/overview. [Kasutatud 10 aprill 2023].

[126] Mozilla Foundation, „Fetch API,“ [Võrgumaterjal]. Kättesaadav: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. [Kasutatud 20 aprill 2023].

[127] Mozilla Foundation, „MDN web docs - Using the Fetch API,“ [Võrgumaterjal]. Kättesaadav: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. [Kasutatud 18 aprill 2023].

[128] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[129] MongoDB, „In-Memory Databases Explained,“ [Võrgumaterjal]. Kättesaadav: https://www.mongodb.com/databases/in-memory-database. [Kasutatud 20 04 2023].

[130] RedHat, „What is Kubernetes?,“ 27 märts 2022. [Võrgumaterjal]. Kättesaadav: https://www.redhat.com/en/topics/containers/what-is-kubernetes.

[131] IBM, „What is Terraform,“ [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/topics/terraform. [Kasutatud 28 aprill 2023].

[132] CLEARVISION, „Compliance Solution for the Medical Devices Industry!,“ [Võrgumaterjal]. Kättesaadav: https://www.clearvision-cm.com/blog/at-last-a-compliance-solution-for-the-medical-devices-industry/. [Kasutatud 10 04 2023].

[133] Appfire, „Comala Document Management,“ Appfire, 2023. [Võrgumaterjal]. Kättesaadav: https://hub.appfire.com/project-and-portfolio-management/comala-document-management/.

[134] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

[135] Tarbijakaitse ja Tehnilise Järelevalve Amet, „CE-märgis,“ [Võrgumaterjal]. Kättesaadav: https://ttja.ee/ariklient/ohutus/ce-margis. [Kasutatud 12 märts 2023].

[136] U.S Food and Drug Administration, „About FDA,“ [Võrgumaterjal]. Kättesaadav: https://www.fda.gov/. [Kasutatud 12 märts 2023].

[137] Internet Engineering Task Force (IETF), „TOTP: Time-Based One-Time Password Algorithm,“ mai 2011. [Võrgumaterjal]. Kättesaadav: https://www.rfc-editor.org/rfc/rfc6238.

[138] A. Rump, „Versions of .NET Core,“ 2023. [Võrgumaterjal]. Kättesaadav: https://versionsof.net/core.

[139] J. Inouye, „WebStorm vs VS Code: Compare top IDEs,“ 09 06 2022. [Võrgumaterjal]. Kättesaadav: https://www.techrepublic.com/article/webstorm-vs-vscode/.

[140] T. Dohmke, „100 million developers and counting,“ 25 jaanuar 2023. [Võrgumaterjal]. Kättesaadav: https://github.blog/2023-01-25-100-million-developers-and-counting/.

[141] GitHub, „GitHub Pricing,“ [Võrgumaterjal]. Kättesaadav: https://github.com/pricing. [Kasutatud 09 aprill 2023].

[142] GitHub, „GitHub Markterplace,“ [Võrgumaterjal]. Kättesaadav: https://github.com/marketplace. [Kasutatud 12 aprill 2023].

[143] K. Yap, „Celebrating 10 million Bitbucket Cloud registered users,“ 17 aprill 2019. [Võrgumaterjal]. Kättesaadav: https://bitbucket.org/blog/celebrating-10-million-bitbucket-cloud-registered-users.

[144] JetBrains, „Webstorm,“ [Võrgumaterjal]. Kättesaadav: https://www.jetbrains.com/webstorm/. [Kasutatud 09 aprill 2023].

[145] Microsoft, „Working with JavaScript,“ [Võrgumaterjal]. Kättesaadav: https://code.visualstudio.com/docs/nodejs/working-with-javascript. [Kasutatud 09 aprill 2023].

[146] JetBrains, „Free Educational Licenses,“ [Võrgumaterjal]. Kättesaadav: https://www.jetbrains.com/community/education/#students. [Kasutatud 09 aprill 2023].

[147] M. Dorrey, „VS Code vs WebStorm: a comparison for newbies,“ 14 märts 2023. [Võrgumaterjal]. Kättesaadav: https://dev.to/markdorrey/vs-code-vs-webstorm-a-comparison-for-newbies-1mi.

[148] Google LLC, „Protocol Buffers Documentation,“ 2023. [Võrgumaterjal]. Kättesaadav: https://protobuf.dev/overview/.

[149] IBM, „Documentantion - What is Ajax?,“ 2 märts 2023. [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=page-asynchronous-javascript-xml-ajax-overview.

[150] RedHat, „What is Infrastructure as Code (IaC)?,“ 11 mai 2022. [Võrgumaterjal]. Kättesaadav: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

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

Mina, Tarvi Raun

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Eesti määrustele vastav kvaliteedijuhtimissüsteemi dokumentide heakskiitmine“, mille juhendaja on German Mumma ning kaasjuhendaja Marion Lepmets
   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.

15.05.2023

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

Lisa 2 – Eksisteerivate lahenduste võrdlustabel

Tabel 12. Eksisteerivate lahenduste võrdlustabel

Toode	**Lühikirjeldus**	**Digiallkirjastamine**	**Hinnastamine**
Qualio	Qualio on kvaliteedijuhtimise platvorm (eQMS), mis võimaldab hallata täielikult kontrollitud töövooge dokumentide kinnitamisteks ja kooskõlastamiseks.	Puudus: elektrooniline allkiri ei vasta eIDAS (910/2014) alusel kvalifitseeritud digiallkirjaks ning vastab ainult *lihtne elektrooniline allkirja (SES)* tasemele.	Puudub selge hinnastamine ning tasuta prooviperiood.
Confluence Cloud	Confluence Cloud on dokumentide haldussüsteem informatsiooni loomiseks, hoiustamiseks, jagamiseks ja koostööks. Platvorm on laiendatav ning kohendatav *plugin*'ite kaudu. Samuti ühildub teiste Atlassiani tootedega. Eestis on laialdaselt kasutusel erinevate valdkondade organisatsioonides kvaliteedijuhtimise (eQMS) süsteemina.	Puudus: Tarkvaral puudub sisse ehitatud võimekus dokumentide elektrooniliseks allkirjastamiseks.	Alates 5 eurot kasutaja kohta kuus. Kuni 10 kasutajaga instants on tasuta.

Toode	Lühikirjeldus	Digiallkirjastamine	Hinnastamine
MediCompli [129]	Tarkvara lahendus, mille eesmärk on abistada meditsiiniseadmete ettevõtetel automatiseerida oma vastavushaldust ja tuua oma meditsiiniseadmed kiiremini turule. Tarkvara on installeeritav ning seadistatav platvormil Confluence Cloud ning laiendab platvormi kasututus võimalusi ja funktsionaalsust.	Puudus: elektrooniline allkiri ei vasta eIDAS (910/2014) alusel kvalifitseeritud digiallkirjaks ning vastab ainult *lihtne elektrooniline allkirja (SES)* tasemele.	Puudub selge hinnastamine ning tasuta prooviperiood.
Comala Document Management [130]	Tarkvara võimaldab meditsiiniseadmete ettevõtetel hallata täielikult kontrollitud töövooge dokumentide kinnitamisteks ja kooskõlastamiseks. Tarkvara on installeeritav ning seadistatav platvormil Confluence Cloud ning laiendab platvormi kasututus võimalusi ja funktsionaalsust.	Puudus: elektrooniline allkiri ei vasta eIDAS (910/2014) alusel kvalifitseeritud digiallkirjaks ning vastab ainult *lihtne elektrooniline allkirja (SES)* tasemele.	Hind sõltub kasutajate arvust. Keskmine hind on 1,5 eurot kasutaja kohta. Lisanduv juurde veel Confluence Cloud tasu.

Lisa 3 - Funktsionaalsete nõuete tulemid koos MoSCoW prioritiseerimisega

Prioritiseeritud nõuded on tabelis märgistatud järgnevate tähistega:

- M - „Peab olema“ ehk ***must have***
- S - „Peaks“ ehk ***should have***
- C - „Võiks“ ehk ***could have***
- W - „Ei tee“ ehk ***won't have***

Tabel 13. Funktsionaalsete nõuete tulemid koos MoSCoW prioritiseerimisega

Kood	**Funktsionaalse nõude kirjeldus**	**MoSCoW**
F1	Atlasssian ID kasutajakontoga sisse loginud kasutaja, kellel on admistraatori õigused, saab Confluence Cloud toote-instantsi (*Confluence instance*) installeerida toote rakendust läbi rakenduste poe Atlasssian Marketplace.	M
F2	Rakenduse installeerimisel lisatakse Confluence Cloud keskkonda kaks uut plugin'it - *Forge Macros* ning *Forge Page Byline*.	M
F3	Atlasssian ID kasutajakontoga saan kasutada Confluence Cloud platvormil toote-instantsi (*Confluence instance*)	
F3	Atlasssian ID kasutajakontoga sisse loginud kasutaja ei pea installeeritud plugin'ite kasutamiseks eraldi sisse logima.	M
F4	Atlasssian ID kasutajakontoga sisse loginud kasutaja saab koheselt kasutada installeeritud rakendusi	
F5	Võimalik on määrada kasutajaid, kellel ei tohi olla volitusi installeeritud lisandmoodulite kasutamiseks.	W
F6	Kõigil sisulehtede tegevuste ribal (*Content byline*) on leitav installeeritud *plugin* (*Forge Page Byline*).	M
F7	Võimalus määrata sisuruume (*Space*), milles installeeritud lisandmoodulid on kasutatavad	C
F8	Sisulehe tegevuste ribal on kuvatud, kas sisulehel on koostatutud allkirjastatud dokument, allkirjastamine on pooleni või kes ei ole allkirjastatud	M

Kood	Funktsionaalse nõude kirjeldus	MoSCoW
F9	Sisulehe tegevuste ribal saavad sisulehe pooleni oleva allkirjastamise andmeid ainult näha sisulehe omanik ning allkirjastajaks määratud kasutajad	S
F10	Sisulehe tegevuste ribal on kuvatud dokumenti allkirjastajad	
F11	Sisulehe tegevuste ribal on võimalus koheselt sisulehest koostada allkirjastada üksikdokument	M
F12	Sisuleht peab olema staatuses, mis lubab allkirjastamist	C
F13	Vastava õigusega kasutaja lisab dokumenti allkirjastamiseks kinnitajaid, kes peavad dokumenti samuti allkirjastama	S
F14	Peale edukat allkirjastamisest märgitakse sisuleht allkirjastatuks	M
F15	Peale edukat allkirjastamisest kõigi kinnitajate poolt märgitakse sisuleht allkirjastatuks	S
F16	Kasutajate allkirjastajaks lisamiseks peab lisataval kasutajal olema volitused dokumentide allkirjastamiseks	S
F17	Sisulehe omanik saab ainult alustada allkirjastamist	M
F18	Sisulehe omaniku on võimalik muuta	C
F19	Ainult sisulehe omanik ning volitatud kasutaja saavad allkirjastamist tühistada	S
F20	Allkirjastajate hilisem juurde lisamine	W
F21	Sisulehe tegevuste ribal on võimalus näha sisulehe allkirjastajaid ning allkirja staatust	S
F22	Sisulehel kuvatakse kasutajale selgelt nähtav sõnum kui sisulehe avab kasutaja, kes on määratud sisulehe kinnitajaks ja allkirjastajaks	S
F23	Kasutaja saab emaili kaudu teavituse, kui ta määratakse kinnitajaks	W
F24	Kasutaja ei suunata allkirjastamisel Confluence keskkonnast välja	M
F25	Sisulehe allkirjastamiseks avatakse kasutajale dialoogaken, kus on võimalik dokument digitaalselt allkirjastada	M
F26	Vahetult enne allkirjastamist eksporditakse sisuleht PDF-formaadi ning valmistakse ette digitaalne allkirjastamise konteiner.	M
F27	Digitaalse allkirjastamise konteinerisse lisatakse samuti sisulehe manused	C

Kood	Funktsionaalse nõude kirjeldus	MoSCoW
F28	Edukalt digitaalselt allkirjastatud konteiner lisatakse sisulehe manuste nimekirja	M
F29	Edukalt digitaalselt allkirjastatud konteineri edastamine emaili kaudu läbi süsteemi	C
F30	Peale edukat allkirjastamisest lisatakse sisulehele uus versiooni number	S
F31	Allkirjastatud sisulehele muutmisel luuakse sisulehest uus mustand	S
F32	Sisulehest uue mustandi loomisel peavad säilima eelnevalt allkirjastatud dokumendid	S
F33	Allkirjastatud sisuleht lukustatakse muutmiseks	W
F34	Sisulehel rakenduste nimekirjast on sisulehele võimalik lisada installeeritud *plugin*'it	S
F35	Sisulehele lisatud *plugin*'i (*Macro*) kuvas kuvatakse välja digitaalse allkirjastamise aeg ja staatus	M
F36	Sisulehele lisatud *plugin*'i (*Macro*) kuvas kuvatakse välja allkirjastajate nimekiri ning allkirja staatus	M
F37	Sisulehele lisatud *plugin*'i (Macro) kuvas kuvatakse välja nimekiri kõik sisulehest koostatud allkirjastatud dokumendid	
F38	Sisulehte eksportimisel PDF-formaadi lisatakse failile *plugin*'i (Macro) kuvatav sisu	S
F39	Kasutajad saavad otsida sisulehti (Content page), mis on allkirjastatud dokumendina	S
F40	Kasutajad saavad otsida sisulehti, kus neid on määratud allkirjastaks	S
F41	Kasutajad saavad otsida sisulehti, kus allkirjastajad on määratud, kuid kõik ei ole veel allkirjastanud	C
F42	Kasutajad saavad otsida sisulehti allkirjastaja alusel	C
F43	Admistraator saab anda kasutajatele allkirjastamise õigust	S
F44	Admistraator saab anda kasutajate õigust allkirjastate lisamiseks	S
F45	Admistraator saab lisada kasutajate gruppe õiguste määramiseks	S

Lisa 4 - Funktsionaalsuse mittefunktsionaalsed nõuded

Tabel 14. Funktsionaalsuse mittefunktsionaalsete nõuete kirjeldus

Kood	**Funktsionaalsuse mittefunktsionaalsete nõuete kirjeldus**
MF-FUN1	Rakendust on võimalik kasutada platvormil Confluence Cloud
MF- FUN2	Lisandmoodulid ei tohi salvestada kasutajate isiklikke andmeid väljaspool Confluence süsteemi
MF- FUN3	Andmebaasi SQL-päringud peavad olema kaitstud SQL-süsti (*SQL-injection*) ründevektorite vastu.
MF- FUN4	Kõik API päringud peavad olema töötama üle HTTPS protokolli ning olema turvatud.
MF- FUN5	Kõik API päringud peavad olema valideeritud vastavalt sisendireeglitele.
MF- FUN6	Andmete edastus peab olema kaitstud kasutades krüpteeritud ja vajadusel autenditud kanalit.
MF- FUN7	API päringud peavad auteriseerimiseks kasutama JWT (*JSON Web Token*) turvaliseks andmeedastuseks
MF- FUN8	Krüptoalgoritmite ja räsifunktsioonide kasutamisel tuleb kasutada turvalisi algoritme ja võtmepikkuseid.
MF- FUN9	Kõik andmed, andmebaasid, SQL skriptid, lähtekood ja rakendus peavad kasutama UTF-8 kodeeringut.

Lisa 5 - Kasutatavuse mittefunktsionaalsed nõuded

Tabel 15. Kasutatavuse mittefunktsionaalsete nõuete kirjeldus

Kood	**Kasutatavuse mittefunktsionaalsete nõuete kirjeldus**
MF-USE1	veebipõhine kasutajaliides peab ühilduma veebilehitsejate Google Chrome, Apple Safari, Mozilla Firefox ning Microsoft Edge viimaste versioonidega ning peab olema kasutatav erinevatelt seadmetelt ja platvormidelt.
MF-USE2	kasutajaliidese disainis tuleb järgida järjepidevust. Ühesuguse funktsiooniga elemendid peavad erinevates moodulites olema visuaalselt identsed ning toimima sama loogikaga.
MF-USE3	kasutajaliideses kasutatav stiil, kirja font ja värvid peavad olema läbivalt ühesugused.
MF-USE4	süsteem peab andma teavitusi andmete salvestamisel, kustutamisel ja vigade korral.
MF-USE5	igale kasutajale kuvatakse menüü vastavalt tema kasutajaõigustele.
MF-USE6	süsteemi poolt saadetavad teavitussõnumid peavad olema selged, lühikesed ja informatiivsed.
MF-USE7	kohustuslike väljade või mittevastava andmetüübi salvestamisel peab süsteem andma veateate ning märkima andmevälja punaseks
MF-USE8	Veebipõhine kasutajaliides peab ühilduma täielikult standarditega HTML 5 ja CSS 3.
MF-USE9	Lisandmoodulid peavad toetama mobiiliseadmete veebilehitsejaid

Lisa 6 - Toetavuse mittefunktsionaalsed nõuded

Tabel 16. Toetavuse mittefunktsionaalsete nõuete kirjeldus

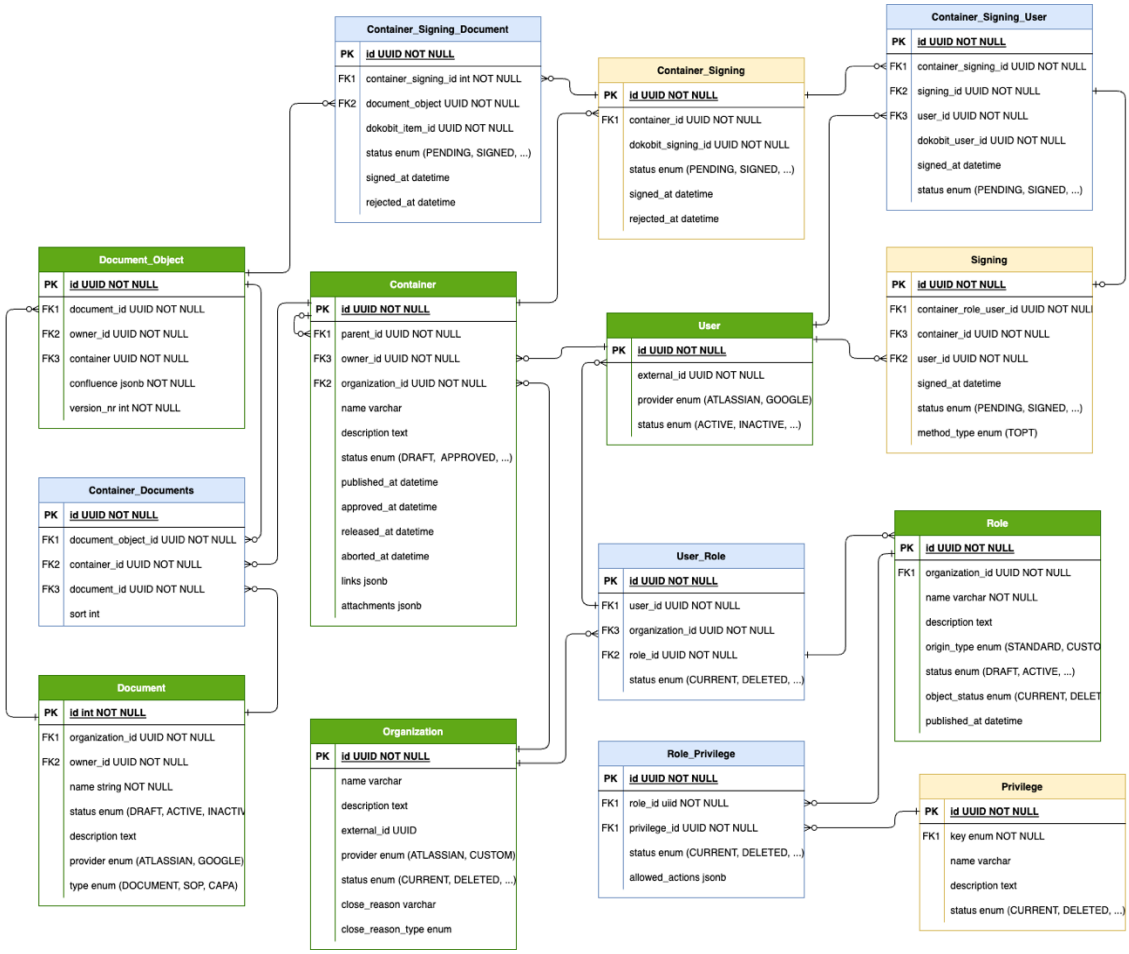
Kood	**Toetavuse mittefunktsionaalsete nõuete kirjeldus**
MF-SUP1	Rakenduse, andmebaasi ja kolmanda osapoole komponendid peavad olema sellised, mille eluea lõpp pole teadaolevalt vähem kui 2 aasta pärast.
MF-SUP2	Rakendust on võimalik liidestada Confluence Cloud API'ga
MF-SUP3	Rakendust on võimalik liidestada Dokobit Gateway API'ga
MF-SUP4	Koodibaasis on unittestidega kaetud 90% funktsionaalsusesest
MF-SUP5	Koodibaasis on integratsioonitestidega kaetud 90% kontrollereid
MF-SUP6	Keskkonnapõhised muutujad peavad olema konfiguratsioonifailist seadistatavad.
MF-SUP7	Rakenduse äriloogika tuleb realiseerida andmebaasist eraldi sõltumatus rakenduskihis
MF-SUP8	Rakenduse uuendustega kaasnevad andmebaasi muudatused tuleb automatiseerida
MF-SUP9	Rakendust peab saama ilma ümberprogrammeerimata liigutada erinevate domeenide ja domeeni saitide vahe
MF-SUP10	Rakendusserveri failisüsteemi ei tohi salvestada midagi püsivaks kasutamiseks.
MF-SUP11	Rakendusserver peab võimaldama töötamist andmebaasiserverist eraldi serveril.
MF-SUP12	Rakendusserver peab olema vajadusel klasterdatav aktiivklastris.
MF-SUP13	Sidusinfosüsteemide mitte kättesaadavus ei tohi segada rakenduse töötamist. Sidusinfosüsteemidega andmevahetamisel tekkinud vead logitakse ja kasutajat hoiatatakse
MF-SUP14	Veebiteenuseid (REST, SOAP) pakkuv rakendus peab olema üles ehitatud nii, et see toetaks teenuste versioneerimist URL-i ja/või skeemi tasemel.

Lisa 7 - Dokumendi koostaja põhised kasutajalood

Tabel 17. Dokumendi koostaja põhised kasutajalood

Kood	**Dokumendi koostaja põhine kasutajalugu**	**Nõue**
UC-K1	Dokumenti koostajana soovin koheselt sisulehest koostada allkirjastatud dokumenti	F11, F14, F24, F25, F26, F30
UC-K2	Dokumenti koostajana soovin sisulehe määrata staatust kui valmis allkirjastamiseks	F12
UC-K3	Dokumenti koostajana soovin lisada dokumenti allkirjastamiseks kinnitajaid, kes peavad samuti dokumenti allkirjastama	F13, F15, F17
UC-K4	Dokumenti koostajana soovin tühistada pooleni oleva dokumenti allkirjastamist	F19
UC-K5	Dokumenti koostajana soovin näha pooleni oleva allkirjastamise allkirjastajate staatust	F21
UC-K6	Dokumenti koostajana soovin allalaadida allkirjastatud digitaalset konteinerit	F28
UC-K7	Dokumenti koostajana soovin juba allkirjastatud sisulehest teha uue mustandi	F31, F32
UC-K8	Dokumenti koostajana soovin lisada nimekirja viimase dokumenti allkirjastajatest	F35, F36, F38
UC-K9	Dokumenti koostajana soovin lisada sisulehel nimekirja kõikist sisulehest koostatud allkirjastatud dokumenditest	F37, F38

Lisa 8 – Andmebaasi loogiline andmemudel



Joonis 8. Andmebaasi olemi-suhte loogiline andmemudel

Lisa 9 – Tagarakenduses kasutatav *Dockerfile*

```
FROM node:14-alpine As development

ARG service

WORKDIR /usr/src/app

COPY package*.json ./

RUN apk add gcc g++ make python3

RUN npm install --only=development

COPY . .

RUN npm run build

FROM node:14-alpine As production

RUN apk add gcc g++ make python3\
   && rm -rf /var/cache/apk/*

ARG NODE_ENV=production
ENV NODE_ENV=${NODE_ENV}

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install --only=production

COPY . .

COPY --from=development /usr/src/app/dist ./dist

ENTRYPOINT [ "/bin/sh" ]
```

Koodinäide 6. Tagarakenduse kasutatav Dockerfile

Lisa 10 – Mikroteenuse *api-gateway* seadistus *docker-compose.yml* failis

```
api-gateway:
    container_name: api-gateway
    build:
      context: .
      dockerfile: Dockerfile.dev
      args:
        service: api-gateway
    command: -c "npm run build -- common && npm run start:dev -- api-gateway"
    ports:
      - "3000:3000"
      - "80:3000"
    environment:
      NODE_ENV: development
      PORT: 3000
      GRPC_AUTH_HOST_URL: "auth-service:50000"
      GRPC_USER_HOST_URL: "user-service:50000"
      GRPC_CONTAINER_HOST_URL: "container-service:50000"
      GRPC_WORKFLOW_HOST_URL: "workflow-service:50000"
      GRPC_ORGANIZATION_HOST_URL: "organization-service:50000"
      GRPC_FIELD_HOST_URL: "field-service:50000"
      GRPC_DOCUMENT_HOST_URL: "document-service:50000"
      GRPC_TASK_HOST_URL: "task-service:50000"
      GRPC_ROLE_HOST_URL: "role-service:50000"
    networks:
      - thesis-network
    depends_on:
      - user-service
      - auth-service
      - container-service
      - workflow-service
      - organization-service
      - field-service
      - document-service
      - task-service
      - role-service

volumes:
    thesis-postgres-db-data:
      driver: local
  networks:
    thesis-network:
      driver: bridge
```

Koodinäide 7. Mikroteenuse *api-gateway* seadistus *docker-compose.yml* failis

Lisa 11 – Andmebaasi migratsioonifaili näide faili 20230414144456_added_role_tables.ts põhjal

```
export async function up(knex: Knex): Promise<void> {
  await knex.schema
    .raw('CREATE EXTENSION IF NOT EXISTS "uuid-ossp"')
    .createTable(DATABASE_TABLE_NAME.ROLE, (table) => {
      table.uuid('id').primary().defaultTo(generateUuidV4(knex));
      table.uuid('organization_id');

      table.string('name', 255).notNullable();
      table.text('description');
      table.timestamp('published_at', { useTz: true });

      // Enumeration fields
      table.enum('status', Object.keys(EBusinessStatus)).notNullable().defaultTo(EBusinessStatus.DRAFT);
      table.enum('origin_type', Object.keys(EOriginType)).notNullable();
      table.enum('module_content_type', Object.keys(EModuleContentType));

      // base meta fields
      addMetaFieldsToMigration(table, knex);
    })
    .createTable(DATABASE_TABLE_NAME.PERMISSION, (table) => {
      table.uuid('id').primary().defaultTo(generateUuidV4(knex));
      table.string('name', 255).notNullable();
      table.text('description');
      table.enum('permission_key', Object.keys(EPermissionKey)).notNullable();

      // base meta fields
      addMetaFieldsToMigration(table, knex);
    })
    .createTable(DATABASE_TABLE_NAME.ROLE_PERMISSION, (table) => {
      table.uuid('id').primary().defaultTo(generateUuidV4(knex));
      table.uuid('role_id').references('id').inTable(DATABASE_TABLE_NAME.ROLE).notNullable().onDelete('CASCADE');
      table
        .uuid('permission_id')
        .references('id')
        .inTable(DATABASE_TABLE_NAME.PERMISSION)
        .notNullable()
        .onDelete('CASCADE');
      table.jsonb('allowed_actions').notNullable().defaultTo(JSON.stringify(defaultPermissionActions));
    });
}

export async function down(knex: Knex): Promise<void> {
  await knex.schema.dropTableIfExists(DATABASE_TABLE_NAME.ROLE_PERMISSION);
  await knex.schema.dropTableIfExists(DATABASE_TABLE_NAME.PERMISSION);
  await knex.schema.dropTableIfExists(DATABASE_TABLE_NAME.ROLE);
}
```

Koodinäide 8. Andmebaasi migratsioonifaili näide faili *20230414144456_added_role_tables.ts* põhjal

Lisa 12 – OpenApi dokumenteerimine kontrollerfailis

```
@Post(':documentId/objects')
@ApiOperation({
  summary: 'Enduser can add new object to document on behalf of their
organization',
  description: 'Current enduser is provided by token payload',
  })
@ApiParam({
  type: String,
  name: 'documentId',
  description: 'Document UUID where to attach object of document'
})
@ApiCreatedResponse({ type: DocumentObjectConfluenceResponseDto })
@ApiBadRequestResponse({
  description: 'Could not find organization by Uuid {organizationId}'
})
@ApiInternalServerErrorResponse({ description: 'Something went wrong' })
@ApiForbiddenResponse({
  description: 'Access denied to document {documentId}'
})
@ApiBody({ type: CreateDocumentObjectConfluenceRequestDto })
async createConfluenceDocumentObject(
  @Param('documentId', ParseUUIDPipe) documentId: Uuid,
  @Body() body: CreateDocumentObjectConfluenceRequestDto,
  @Token() token: string
): Promise<DocumentObjectConfluenceResponseDto> {
```

Koodinäide 9. OpenApi kirjelduste määramine kontrollerfailis *document-confluence.controller.ts*

Lisa 13 – Täiendav konfiguratsioonifail *configuration.ts*

```
export default () => ({
  environment: process.env.NODE_ENV,
  port: parseInt(process.env.PORT, 10) || 3000,
  database: {
    psql: {
      host: process.env.POSTGRES_HOST,
      port: parseInt(process.env.POSTGRES_PORT, 10) || 5432,
      database: process.env.POSTGRES_NAME,
      username: process.env.POSTGRES_USERNAME,
      password: process.env.POSTGRES_PASSWORD,
    },
  },
  jwt: {
    appName: process.env.TWO_FACTOR_AUTHENTICATION_APP_NAME,
    regular: {
      accessTokenSecret: process.env.ACCESS_TOKEN_SECRET,
      refreshTokenSecret: process.env.REFRESH_TOKEN_SECRET,
    },
    system: {
      accessTokenSecret: process.env.ACCESS_TOKEN_SECRET_SYSTEM,
      refreshTokenSecret: process.env.REFRESH_TOKEN_SECRET_SYSTEM,
    },
    accessTokenExpiresIn: process.env.ACCESS_TOKEN_EXPIRES_IN,
    refreshTokenExpiresIn: process.env.REFRESH_TOKEN_EXPIRES_IN,
  },
  dokobit: {
    baseUrl: process.env.DOKOBIT_BASE_URL,
    accessToken: process.env.DOKOBIT_ACCESS_TOKEN,
  },
  cors: {
    origin: process.env.CORS_ORIGIN,
  },
});
```

Koodinäide 10. Täiendav konfiguratsioonifail configuration.ts

Lisa 14 – Valideerimisklassis *create-document-request.dto.ts*

```
export class CreateDocumentConfluenceRequestDto {
  @ApiProperty({ required: true, type: 'string', format: 'uuid' })
  @IsNotEmpty()
  @IsUUID()
  organizationId: Uuid;

  @ApiProperty({ required: true })
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  @MaxLength(100)
  title: string;

  @ApiPropertyOptional()
  @IsString()
  @IsOptional()
  @MaxLength(1000)
  description?: string;

  @ApiProperty({ required: true, enum: EDocumentType })
  @IsNotEmpty()
  @IsEnum(EDocumentType)
  documentType: EDocumentType;

  @ApiProperty({ required: true, type: 'number' })
  @IsNotEmpty()
  @IsInt()
  @Min(1)
  startVersioningAt: number;

  @ApiProperty({ type: ConfluenceAttributesDto })
  @IsObject()
  @ValidateNested()
  @Type(() => ConfluenceAttributesDto)
  confluenceAttributes: ConfluenceAttributesDto;
}
```

Koodinäide 11. Valideerimisklassi näide faili *create-document-request.dto.ts* põhjal

Lisa 15 – Eesrakenduses kasutatavad abimeetodid API'ga suhtlemiseks

```
// useFetch.ts
export const useFetch = ({
  key, url,
  method = EMethod.GET,
  keepPreviousData = false
}: IQueryParams) => {
  const { data: env } = useViewEnv();
  const { data, ...rest } = useQuery({
    queryKey: key,
    queryFn: async () => {
      if (env?.backendHost) {
        const tokenResponse = await getToken(env.backendHost);
        const jtwData: IJwt = tokenResponse.bodyResponse;
        return await externalApiCall({
          url: env.backendHost + http.BACKEND_VERSION_PATH + url,
          token: jtwData.accessToken,
          method: method,
        });
      }
      throw new Error('Env data not retrieved');
    },
    keepPreviousData: keepPreviousData,
  });

  return { data, ...rest };
};

// documentList.ts
const { data: documents } = useFetch({
  key: [EDocumentQueryKeys.DOCUMENT_KEY, EBusinessStatus.ACTIVE],
  url: `${http.DOCUMENT_CONFLUENCE_API_PATH}?${buildQuery()}`,
  method: EMethod.GET,
});
```

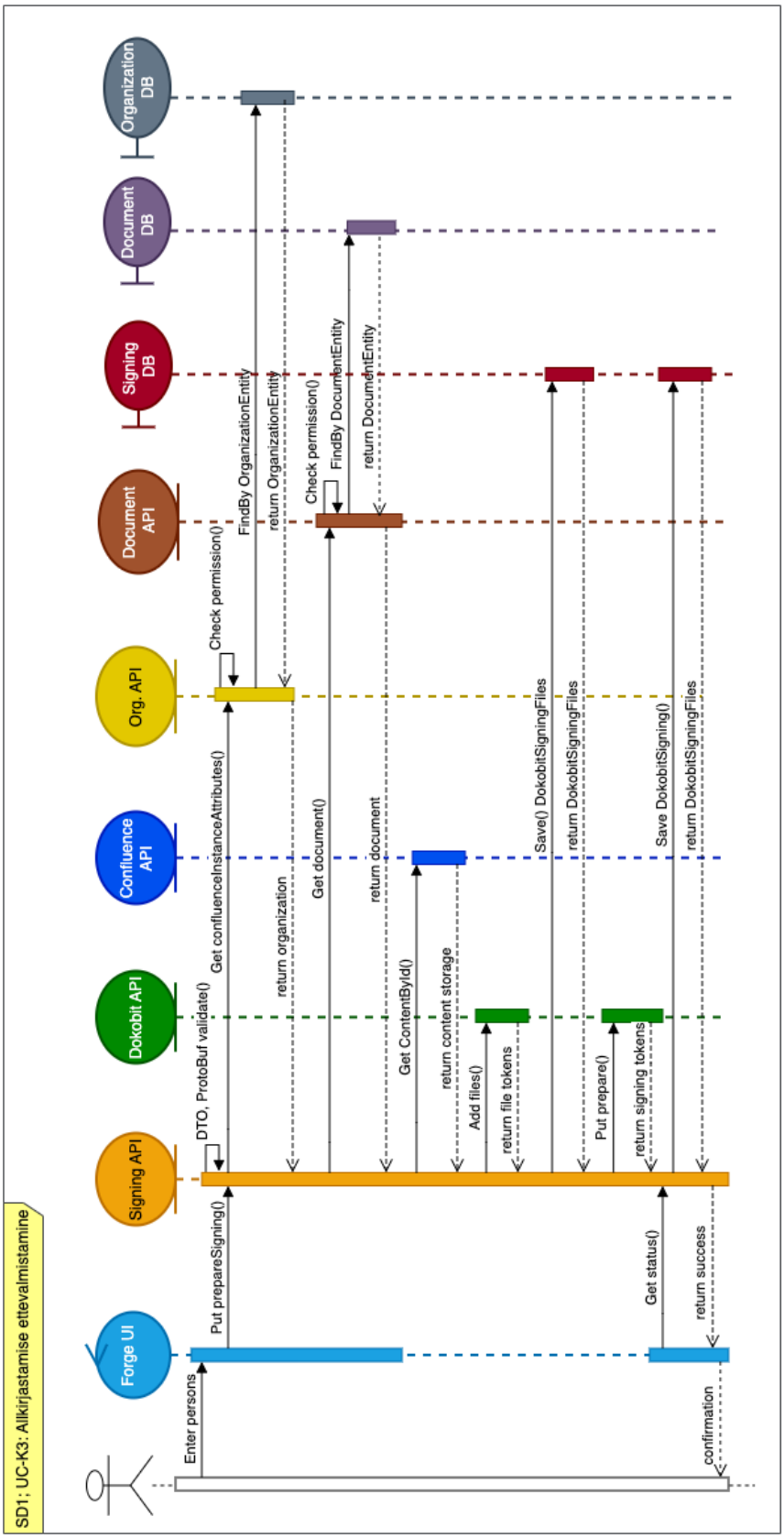
Koodinäide 12. Eesrakenduses kasutatavad abimeetodid API'ga suhtlemiseks

Lisa 16 – Automaattesti faili *document-service.spec.ts*

```
describe('Create new document object (confluence)', () => {
  beforeEach(async () => {
    await repositoryDocument.save(mockDocumentList);
  });

  it('should throw CustomNotFoundRPCException if owner is not found', async () => {
    const creationData = { ...mockRequest, ownerId: randomId };
    await expect(
      service.createConfluenceDocumentObject(mockDoc.id, creationData, authorizedUser, metadata)
    ).rejects.toThrowError(
      new CustomNotFoundRPCException(`Could not find active user by Uuid ${creationData.ownerId}`)
    );
  });

  it('should throw CustomNotFoundRPCException if document to attach is not found', async () => {
    await expect(
      service.createConfluenceDocumentObject(randomId, mockRequest, authorizedUser, metadata)
    ).rejects.toThrowError(
      new CustomNotFoundRPCException(`Could not find document with UUID ${randomId}`)
    );
  });

  it('should create and save new document object', async () => {
    const documentId = mockDocumentConfluence.id;
    const documentObject = await service.createConfluenceDocumentObject(
      documentId,
      mockDocumentObjectConfluenceRequest,
      authorizedUser,
      metadata
    );

    expect(documentObject).toBeObject();
    expect(documentObject).toContainAllKeys(Object.keys(mockDocumentObjectResponse));
    expect(documentObject).toMatchObject({
      ...mockDocumentObjectResponse,
      id: documentObject.id,
    });
  });

  it('should call AbstractDocumentService method "checkDocumentAccess"', async () => {
    const documentId = mockDocumentConfluence.id;
    const checkDocumentAccessFn = jest.spyOn(service, 'checkDocumentAccess');

    await service.createConfluenceDocumentObject(
      documentId,
      mockDocumentObjectConfluenceRequest,
      authorizedUser,
      metadata
    );

    expect(checkDocumentAccessFn).toHaveBeenCalledTimes(1);
    expect(checkDocumentAccessFn).toHaveBeenCalledWith(mockDocumentConfluence, authorizedUser);
  });
});
```

Koodinäide 13. automaattest faili põhjal document-service.spec.ts, mis kontrollib teenuse võimekust visata erandeid oodatud viisil ning tagastusväärtusi

Lisa 17 – Järgnevusdiagramm kasutajalugu UC-K3 põhjal



Joonis 9. Järgnevusdiagramm kasutajalugu UC-K3 põhjal

Lisa 18 – REST API dokumentatsioon

API dokumentatsioon – http://api.tarviraun.eu/api

Postman'i veebiplatvormil ettevalmistatud API-teekonnad – https://www.postman.com/interstellar-escape-679558/workspace/final-thesis/documentation/7055743-e7895a86-3ff7-4951-ac9a-73fb962f7830