TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Sergei Pojev 176294

Paberivaba klienditeenindus pangas

Bakalaureusetöö

Juhendaja: Meelis Antoi

Magistrikraad

Tallinn 2021

Autorideklaratsioon

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

Autor: Sergei Pojev

05.01.2021

Annotatsioon

Bakalaureusetöö eesmärk on pakkuda välja idee ja tehnoloogiline lahendus jätkusuutliku ja kaasaegse klienditeenindusprotsessi loomiseks finantsettevõttes. Lahenduse analüüsis tuuakse välja paberivaba teeninduse kasutegurid organisatsiooni majandustegevusele ja keskkonnale. Töös kirjeldatakse ärilisi tingimusi, mis peavad olema täidetud tehnoloogilise lahenduse kasutuselevõtuks.

Tehnoloogia lahenduse kirjelduses tuuakse välja võimalused sellise lahenduse loomiseks. Töös on välja toodud võimalik süsteemi arhitektuur, seosed erinevate komponentide vahel ja tehnoloogiad, mida kasutatakse paberivaba lahenduse loomiseks.

Arendusprotsessi käigus on näidatud, kuidas on implementeeritud põhilised programmi osad. Põhirõhk on kliendi- ja serverirakenduse ühendamisel ning kasutataval tehnoloogial. Bakalaureusetöö tulemusena loodi kontseptsiooni tõestuse *(POC)* koos klienditeenindusprotsessi läbiviimiseks vajalike komponentidega. Töös oli kasutatud kaasaegseid tehnoloogilisi lahendusi, mis võimaldavad rakendust tulevikus edasi arendada ja ühendada teiste süsteemidega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 48 leheküljel, 5 peatükki, 20 joonist, 1 tabeli.

Abstract

Paperfree Customer Service in Bank

The aim of this bachelor thesis is to offer an idea and technological solution for creating a sustainable and modern customer service process in a financial company. In the course of the analysis of the solution, the benefits of using paperfree customer service for companies economical activities and natural environment were pointed out. The thesis describes business requirements that must be met in order to start using paperfree customer service.

In the description of solution, the possibilities for creating this program are outlined. Thesis provides possible architectural view, the relationships between various components and the technologies used to create the paperfree service.

The development process part shows how the main parts of the program have been implemented. The main emphasis is on connecting the client application and the server side application and the technological solution used.

As a result of the bachelor's thesis, a proof of concept *(POC)* was created, with the components needed to carry out the customer service process. In this thesis, modern technological solutions were used, that will allow this application to be further developed and integrated with other systems.

The thesis is in Estonian and contains 48 pages of text, 5 chapters, 20 figures, 1 table.

Lühendite ja mõistete sõnastik

API	Application Programming Interface (rakendustarkvara liides)
CI / CD	Continuous Integration and Continuous Delivery - pidev integreerimine ja pidev edastamine.
CLI	Command Line Interface - käsuprotsessor
CSS	Cascading Style Sheets - kaskaadlaadistik
DevOps	Development and Operations (arendus ja käitus) - kultuur, mis edendab käituse -ja arendusmeeskonna koostööd [2].
DOM	Document Object Model - dokumendi objektimudel
Enum	Java keele klass, mis representeerib konstantide gruppi [3]
HTML	Hypertext Markup Language (hüperteksti märgistuskeel)
IKT	Info- ja kommunikatsioonitehnoloogia
IT	Infotehnoloogia
Jira	Scrum ja Kanban meetoditel põhinev agiilse projektijuhtimise vahend, mis sobib erinevatele meeskondadele [1]
JSON	JavaScript Object Notation
NoSQL	Non-SQL, non-relational – mitterelatsiooniline andmebaas
NPM	Node Package Manager
PHP	Hypertext Preprocessor, skriptimiskeel
POC	Proof on concept
POM	Project Object Model
SSL	Secure Socket Layer (turvasoklite kiht, infoturbe protokoll)
STOMP	The Simple Text Oriented Messaging Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Sisukord

Autorideklaratsioon ... 2

Annotatsioon ... 3

Abstract Paperfree customer service in banks ... 4

Lühendite ja mõistete sõnastik ... 5

Sisukord ... 6

Jooniste loetelu ... 8

Tabelite loetelu ... 10

1 Sissejuhatus ... 11

2 Probleemi kirjeldus ... 13

- 2.1 Paberivaba klienditeenindus ... 15
- 2.2 Lahenduse eeltingimused ... 15
- 2.3 Lahenduse skoop ... 17
- 2.4 Sarnased lahendused ... 18
- 2.5 Võimalikud kasutusriigid (turg) ... 19

3 Paberivaba klienditeeninduse lahenduse kirjeldus ... 21

- 3.1 Lahenduse funktsionaalsed nõuded ... 21
- 3.2 Lahenduse mittefunktsionaalsed nõuded ... 22
- 3.3 Tehnoloogia valik ... 23
  - 3.3.1 Serverirakenduse arenduskeelte alternatiivid ... 23
  - 3.3.2 Serverirakenduse raamistiku valimine ... 25
  - 3.3.3 Kasutajaliidese raamistiku valimine ... 26
  - 3.3.4 Andmebaasi valimine ... 28
  - 3.3.5 Versioonihalduskeskkonna valik ... 30
  - 3.3.6 Arenduskeskkonna valik ... 31
- 3.4 Lahenduse arhitektuur ... 32
  - 3.4.1 Protsessi kirjeldus ... 34

3.4.2 Andmete voog ... 35
3.4.3 Andmebaasi mudel ... 36
3.4.4 Võimalik arhitektuuriline optimeerimine ... 37
3.5 Analüüsi kokkuvõtte ... 38
4 Lahenduse arendus ... 39
4.1 Serveri rakenduse arendus ... 39
4.1.1 Spring Boot rakenduse loomine ... 39
4.1.2 Programmi struktuur ... 41
4.1.3 Kasutatavad sõltuvused ... 42
4.1.4 Serverrakenduse konfiguratsioon ... 44
4.1.5 REST API ... 45
4.1.6 Ühendus andmebaasiga ... 46
4.1.7 WebSocketi kasutamine ... 47
4.2 Kliendi rakenduse arendus ... 48
4.2.1 Angulari rakenduse loomine ... 48
4.2.2 Programmi struktuur ... 50
4.2.3 Päringute tegemine vastu serverrakendust ... 52
4.2.4 WebSocketi kuulaja kliendirakenduses ... 53
4.2.5 Kasutajaliidese välimus ... 54
4.3 Testimine ... 55
5 Kokkuvõte ... 58
Kasutatud kirjandus ... 59
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks ... 70
Lisa 2 – Serverirakenduse POM faili sisu ... 71
Lisa 3 – Retrofit konfiguratsioon ... 73
Lisa 4 – Serverrakenduse sõnumite saatmise teenuse implementatsioon ... 74
Lisa 5 – Kliendirakenduse HTTP päringute baasteenus ... 77

Jooniste loetelu

Joonis 1. Paberivaba lahenduse arhitektuur. 33

Joonis 2. Andmete voog. 35

Joonis 3. Kliendi arvuti ja nõustaja arvuti kaardistuse andmebaasi mudel. 36

Joonis 4. Kliendi arvuti reklaami salvestamise andmebaasi mudel. 37

Joonis 5. Projekti loomine IntelliJ IDEA keskkonnas. 40

Joonis 6. Serverrakenduse kihtide mudel. 41

Joonis 7. Maven sõltuvuste lisamine. 43

Joonis 8. Retrofit konfiguratsioon. 44

Joonis 9. WebSocket konfiguratsioon. 45

Joonis 10. Andmebaasi ühenduse loomine. 47

Joonis 11. Objekti süstimine teenusesse. 47

Joonis 12. Sõnumi saatmine läbi WebSocket. 48

Joonis 13. Angular CLI installeerimine. 49

Joonis 14. Uue projekti loomine kasutades Angular CLI-d. 49

Joonis 15. Kliendirakenduse komponentide jaotus ja nende seosed. 50

Joonis 16. Komponentide, teenuste loomine ja programmi käivitamine kasutades Angular CLI-d. 51

Joonis 17. HTTP päringu tegemine Angularis. 52

Joonis 18. WebSocketi konfiguratsioon kliendirakenduses. 53

Joonis 19. Sõnumite kuulamise näide. 54

Joonis 20. Kliendi autentimise vaade. 55

Tabelite loetelu

Tabel 1. REST API kirjeldus. 46

1 Sissejuhatus

Jätkusuutlik areng muutub kõigi ettevõtete jaoks aina olulisemaks ning edasiliikumine selles valdkonnas on väga kiire. Seepärast on paljud ettevõtted otsustanud investeerida tehnoloogiatesse, mis võiksid vähendada saastet.

Üks ettevõtlussektor, kus tegeletakse erinevate jätkusuutlike suundade leidmisega, on pangandus. Panganduses on oluline roll klienditeenindusel, mistõttu on oluline mõelda, kuidas saaks muuta klienditeenindusprotsessi keskkonnasõbralikumaks. Võimalus on näiteks proovida säästa paberit, mida prinditakse kliendiga lepingute allkirjastamiseks. Iga leping prinditakse kahes eksemplaris, millest üks kuulub kliendile ja teine jääb panka.

Muutes klienditeeninduse paberivabaks, säästame loodust tavapärase paberikulu arvelt. Üks Eestis tegutsev pank võib päevas printida umbes 10970 lehte, mis teeb kokku 22 paberipakki. Kasutades meie ja ka naaberriikide võimalusi digitaalseks allkirjastamiseks, suudame seda arvu oluliselt vähendada. Sellega omakorda säästame loodust ja toome turule jätkusuutliku lahenduse tulevastele põlvkondadele.

Vaatamata sellele, et paberivaba kontori lahendus säästab loodust, toob see ka ettevõttele majandusliku kasu. Paberi sisseostu, printeritooneri ja printerimasinate kulud vähenevad. Samuti kaob vajadus ladustada ja arhiveerida paberil prinditud ja allkirjastatud lepinguid, mis on suur kuluallikas igale klienditeenindusega tegelevale ettevõttele. Väheneb ka risk lepinguid kaotada või hävitada, mis on oluline nii ettevõttele kui ka aitab luua klientides usaldust ettevõtte vastu.

Kliendi poolt analüüsides selgub, et ka neile on see kasulik. Kaob vajadus hoida allkirjastatud lepinguid oma kodus ning lepingud on lihtsasti igal ajal kättesaadavad internetipangas või leitavad oma e-postis.

Selle lõputöö eesmärk on tõestada, et on võimalik arendada valmis lahendus, mis pakuks pankadele uut viisi klienditeenindust, kus lepinguid saab kliendiga allkirjastada ja säilitada digitaalselt, kasutades selleks ID-kaarti, Mobiil-ID ja Smart-ID võimalust. Eesmärk on ka pakkuda välja tehnoloogiaid, kuidas seda ideed on võimalik realiseerida.

Tehnoloogiate valikul on autori eesmärk keskenduda nende lihtsusele, piisava dokumentatsiooni olemasolule ning suurele kasutajabaasile.

Töö koosneb kolmest osast. Töö esimeses osas on probleemi kirjeldus. Teine osa koosneb põhjalikust lahenduse analüüsist ja tehnoloogiate valikust. Töö kolmandas, praktilises osas näitan ja kirjeldan põhifunktsionaalsuse osi.

2 Probleemi kirjeldus

Pangandus on üks suuremaid valdkondi, mis on otseselt seotud inimeste teenindamisega. Nii Eestis kui ka teistes riikides on pankadel kliendikeskused, kuhu inimesed võivad pöörduda. Nende külastus on vaatamata kõikidele tehnoloogilistele võimalustele (internetipank, mobiilirakendus, videonõustamised) veel suur. Ehkki pangad soovivad vähendada oma kontorite külastust pakkudes välja erinevaid internetilahendusi, tuleb siiski arvestada, et mõned inimesed eelistavad kohtuda nõustajaga kontoris, et arutada ja leida lahendus oma probleemidele.

Analüüsides eeltoodut, võib järeldada, et panga jätkusuutlikust ei saa tagada ainult internetis pakutavate lahendustega. Tuleb mõelda ka võimalustele, kuidas muuta paremaks ja mugavamaks kliendikogemust pangakontoris. Erinevus internetis sõlmitud ja kontoris vormistatud lepingute vahel on see, et kontorist lahkudes võtab klient endaga kaasa paberile prinditud dokumendid. Üks võimalus on muuta seda kogemust ja pakkuda kliendile teenindust, millega kõik tema lepingud oleksid vormistatud digitaalselt ning lihtsasti igal ajal internetipangast kättesaadavad.

Lisaks paberil dokumentidele võib kliendil olla ebamugav suhelda klienditeenindajaga, kui ta ei näe informatsiooni, mida teenindaja parasjagu näeb. Oleks hea, kui klient saaks jälgida, mida teenindaja parasjagu tema kohta näeb ning klient saaks kontrollida andmete õigsust, ilma et ta peaks hiljem klienditeenindajat parandama ja protsessi uuesti alustama. Klient tahab operatiivselt saada informatsiooni nii oma andmete kui ka võimalike pakkumiste kohta, ilma et teenindaja peaks eraldi seda talle ütlema. See muudab kogemuse paremaks nii kliendi kui ka klienditeenindaja jaoks.

Kliendi mugavuse ja teenindusprotsessi läbipaistvuse kõrval leidub ka probleeme, mis tekib juhul, kui klient külastab pangakontorit. Näiteks on oluline risk seotud lepingute hävimise või kadumisega. Nii klient kui ka pank võib kaotada lepinguid, mis vormistati kontorikülastuse ajal. Kliendi poolelt vaadates võime näha, et inimene lahkub paberile prinditud dokumentidega. Tekib risk dokumente kaotada, mis tähendab, et inimese personaalne informatsioon võib sattuda valedesse kätesse. Kui dokumente peaks hiljem

vaja olema, peab klient tegema pangale päringu, et pank saadaks sõlmitud lepingute koopiad. Koopiate pärimine pangast on kliendile tasuline teenus, sest see on pikk protsess, kus tuleb vajalikud dokumendid panga arhiivist leida ja nende koopiad kliendile saata. Seega näeme, et paberile prinditud lepingud ei ole kõige parem lahendus tänapäeva tehnoloogiliste võimaluste olemasolul.

Pangal on samasugune risk dokumente kaotada. Iga klienditeenindaja vastutab dokumentide säilitamise eest enne, kui need on üle antud arhiivi. Töötajatel on suur vastutus nii panga kui ka klientide ees, et midagi ei läheks kaduma või dokumendid ei satuks valedesse kätesse. Nimetatud riskide kõrval kaasneb paberdokumentidega ka rahaline kulu. Pangale tekib lepingu originaalide säilitamise kulu. Vastavalt Eesti riigi seadustele peab ettevõte säilitama dokumente umbes 8 kuni 10 aastat [4], vastavalt dokumendi tüübile. Siit võib järeldada, kui palju füüsilist ruumi on vaja, et arhiveerida igapäevases klienditeenindusprotsessis sõlmitud paberdokumente.

Füüsilise ruumi rentimise kõrval on ka teised kulutused, mis kaasnevad dokumentide säilitamisega. Esiteks peab dokumendid turvaliselt arhiivi transportima. Oluline on tagada arhiivi turvalisus, et kõrvalised isikud ei pääseks dokumentidele ligi. Peale dokumendi säilitusaja nõude lõppu tuleb dokumendid hävitada. Arhiivist andmete leidmine on keeruline ning tuleb pidada registrit, et dokumendid oleksid igal ajal kättesaadavad. Ehk siis pangale kaasneb erinevaid kulusid, mis on seotud dokumentide hoidmise ja töötlemisega.

Lisaks ärilistele riskidele on aktuaalseks teemaks keskkonnaprobleemid. Iga lepingu sõlmimisel pangakontoris prinditakse välja hulk pabereid. Nende arv sõltub lepingu tüübist. Lisaks lepingule prinditakse panga- ja tootetingimusi. Arvestama peab sellega, et klient peab füüsiliselt allkirja andma, mistõttu peab igast dokumendist tegema kaks koopiat. Üks Eestis tegutsev pank võib päevas printida umbes 10970 lehte, mis teeb kokku 22 paberipakki [5]. Paberi peamine komponent on tselluloos, mis saadakse kuuse, männi või kasepuidust. Seega võib järeldada, et antud koguse paberi printimiseks kulub umbes 1,32 puud arvestades, et ühest keskmisest puust saab umbes 8333 printerilehte [6]. See on märkimisväärne kogus puid mis kulub, et katta ainult ühe panga kontorite vajadust ühes riigis.

Paberil lepingute vormistamine toob endaga kaasa nii riske kliendile kui ka pangale, on majanduslikult kulukas ning mõjutab looduskeskkonda. Lahendus ei ole jätkusuutlik ega mugav. Pankade eesmärk peaks olema seejuures keskonnasõbraliku teenindusprotsessi loomine.

2.1 Paberivaba klienditeenindus

Eelnimetatud probleemide lahendamiseks tuleb pöörduda tehnoloogiliste võimaluste poole. Analüüsides probleeme ja riske järeldasin, et üks võimalikest lahendustest oleks paberivaba klienditeenindus.

Lahenduse peamine idee seisneb selles, et klienditeeninduse ajal ei ole lepingute vormistamisel vaja printida ühtegi paberit. Kõik toimingud ning nii kliendi kui ka panga allkiri oleks digitaalne. Kõik dokumendid oleks hoiustatud digitaalselt ning igal ajal nii kliendile kui ka pangale kättesaadavad.

Paberivaba teenindus on mugav, sest lahendus võimaldab näidata kliendile reaalajas, mida klienditeenindaja parasjagu näeb või süsteemi sisestab. Enne lepingu sõlmimist saab klient üle vaadata oma andmed ja vajadusel neid parandada. Kui kõik sobib, annab klient oma allkirja kasutades digiallkirja võimalusi. Teiselt poolt allkirjastab dokumendi ka panganõustaja, mille järel dokumendid salvestatakse panga andmebaasidesse. Hiljem saab klient kõik oma digidokumendid üle vaadata internetipangas. Kui kliendil puudub internetipanga leping, on võimalik dokument tellida endale e-posti aadressile.

Paberivaba teeninduse lahendus on jätkusuutlik, sest parandab kasutajakogemust, on mugav nii klientidele kui ka töötajatele, ei vaja arhiveerimise kulutusi ning hoiab ära riske, mis on seotud dokumentide kadumisega. Samal aja säästame ka loodust, hoides igapäevaselt hulga pabereid kokku.

2.2 Lahenduse eeltingimused

Selleks, et kirjeldatud lahendust saaks pangas kasutada, peavad olema täidetud mõned eeltingimused. Kõige tähtsam nendest tingimustest on digiallkirja olemasolu ning selle juriidiline kehtivus riiklikul tasemel.

Eestis on kasutusele võetud „Digiallkirja seadus“ 8. märtsil 2000. aastal [7]. Seadus sätestab digitaalallkirja kasutamiseks vajalikud tingimused ning sertifitseerimisteenuse ja ajatempliteenuse osutamise üle järelevalve teostamise korra [7]. Teisisõnu võimaldab see seadus kasutada digitaalselt antud allkirju selliselt, et allkiri omab täpselt sama juriidilist jõudu kui paberile käsitsi antud allkiri. Seadus määrab ka järgmised nõuded, millele peab digiallkiri vastama. Digiallkiri peab [7]:

- Võimaldama üheselt tuvastada isiku, kelle nimel allkiri on antud.
- Võimaldama kindlaks teha allkirja andmise aja.
- Siduma digitaalallkirja andmetega sellisel viisil, mis välistab võimaluse tuvastamata muuta andmeid või nende tähendust pärast allkirja andmist.

Sellega seoses tekib eeltingimus, mille kohaselt riigis peab olema võimalus anda digiallkirju. Paberivaba teeninduse lahendus kasutab digiallkirja võimalusi, et dokumente paberivabalt allkirjastada.

Pangas peab olema ka arendatud süsteem, mis võimaldab klientide digitaalset autentimist ja digiallkirja andmist. Vajalik on tsentraliseeritud süsteem, millega on võimalik integreerida ja süsteemi kasutades peab olema võimalik klienti digitaalselt autentida ehk tema isikut tuvastada. Eestis on näiteks sellisteks võimalusteks ID-kaart, Mobiil-ID ja Smart-ID.

Järgmiseks on vaja teenust, mis võimaldab allkirju anda. Kui klient või klienditeenindaja tahab lepingut või muud dokumenti allkirjastada, peab olema süsteem, mis oskab seda teha. Paberivaba lahendus võimaldab teenusele kõik vajaliku informatsiooni (isikukood, sertifikaadid, jne), mille tulemusel peab valmima digiümbrik. See on fail, kuhu kogutakse digitaalset allkirjastamist vajavad dokumendid ja allkirjastatakse (st varustatakse digitaalallkirja ja kehtivuskinnitusega). Ümbrik sisaldab üht või mitut dokumenti ja ühe või mitme isiku digitaalallkirja koos vastavate kehtivuskinnitustega [8]. Eelduseks on, et digiümbrik on panga andmebaasides salvestatud ning seostatud nii lepingu kui ka kliendiga. Vajadusel peab teenus olema suuteline leidma ümbriku kas kliendi, klienditeenindaja või lepingu põhjal.

Eeldades, et panga tegutsemise riigis on seadusega lubatud kasutada digiallkirjastamist ja et pangas on juba arendatud süsteem, mis selliseid allkirju oskab anda, võib vaadelda panka kui potentsiaalset lahenduse kasutajat. Tuleb aga kindlaks teha, kas vastaval pangal on olemas ka pangakontorid, sest tänapäeval leidub juba ainult internetikeskkonnas tegusevaid panku. Lisaks peab pangas olema arendatud süsteem, millega töötavad klienditeenindajad. Just see süsteem hakkab paberivaba lahendusega koostööd tegema, pakkudes kliendile paberivaba teenindust.

Vajalik on võimalus paigaldada klienditeenindaja töölauale veel üks arvuti kliendi jaoks. See on arvuti, kust klient hakkab nägema informatsiooni ning allkirjastama lepinguid. Eestis peaks arvutil olema kaardilugeja ja võimalus paigaldada arvutile Eesti ID-tarkvara, et saaks autentimiseks ja allkirjastamiseks kasutada ID-kaarti. Kindlasti peab kliendile mõeldud arvutil olema võrguühendus. Kliendi arvutiks võib näiteks kasutada tahvelarvutit.

Kokkuvõtteks võib öelda, et paberivaba lahenduse kasutuselevõtuks peavad olema täidetud päris mitu eeltingimust: olemasolevad pangakontorid, pank peab toetama digitaalset autentimist ja allkirjastamist ning riik peab tunnistama digitaalselt loodud konteinereid ja allkirju.

2.3 Lahenduse skoop

Paberivaba lahendus on suur süsteem, milles peab eksisteerima mitu teenust, et lahendust saaks vastavalt nõuetele kasutada. Seepärast tuleb defineerida skoop.

Suur osa paberivabast lahendusest on kliendi autentimine ja allkirjade andmine. See eeldab programmiosade arendamist, mis suhtlevad otse riigi poolt pakutavate teenustega. Eestis arendab selliseid teenuseid SK ID Solutions AS (varem sertifitseerimiskeskus), mis omab API ning mille teenuseid kasutades on võimalik anda allkirju. Teiseks peab olemas olema andmebaas, kus hoitakse allkirjastatud digiümbrikuid ning pakutakse teenust nende otsimiseks ja tagastamiseks teistele programmidele. Need süsteemid ei kuulu paberivaba lahenduse skoopi, kuna eeldame, et sellised süsteemid on juba pangas arendatud. Kõik tähtsamad Eesti, Läti ja Leedu pangad oskavad juba klienti autentida ning allkirjastada lepinguid või anda maksekinnitusi. Seda tehakse näiteks internetipanka

kasutades. Seega võib öelda, et pankades juba eksisteerivad süsteemid ning neid on võimalik kasutada ka paberivaba lahenduse heaks.

Paberivaba lahendus ei tegele allkirjastamise kasutajaliidese arendamisega. Eeldatud on seda, et kui hakatakse lepingut allkirjastama, siis saadetakse kogu vajalik informatsioon allkirjastamise süsteemile ning tehakse ümbersuunamine allkirjastamise kasutajaliidesele, kus klient saab valida allkirjastamise meetodi ja vajalikud toimingud teha. Kui allkirjad on kogutud, saadetakse paberivaba lahendusele teatis, et toimingud on lõpetatud ning paberivaba lahendus saab jätkata oma tööga.

Kirjeldatava lahenduse skoobis on kahe suurema süsteemi arendus. Esimeseks on paberivaba serverirakenduse (API) ja kasutajaliidese loomine, mis hakkab tööle kliendi arvutis (näiteks tahvelarvutis). Serverirakendus pakub teistele pangasüsteemidele API. Integraatorid hakkavad kasutama programmiliidest, et saata informatsiooni, mis on parajasti vaja kliendile näidata. Samuti saadetakse käsud nagu allkirjastamise käsk või teeninduse lõpetamise käsk. Koos API arendusega seostatakse sellega ka andmebaas, mis on vajalik paberivaba lahenduste seadistuste hoidmiseks.

Skoobis on ka kasutajaliidese arendus, mis töötab kliendi arvutis. See osa tegeleb informatsiooni ja reklaampiltide kuvamisega. Kasutajaliides ja serverirakendus töötavad koos, et võimaldada kasutajale paberivaba teenindust.

Ehkki allkirjade andmine ei ole skoobis, hoolitseb paberivaba lahendus selle eest, et õigesti, korrektselt ja turvaliselt kliendile informatsiooni näidata ning teda allkirjastamise lehele suunata.

2.4 Sarnased lahendused

Alates 2013 aastast kasutab sarnast lahendust SEB Pank AS [9]. Ettevõte nimetab oma programmi Paberivabaks teeninduseks. See võimaldab teeninduse käigus kõik dokumendid digitaalselt allkirjastada ning hiljem on need leitavad kliendile tema internetipangas. See on seni ainus pank Balti riikides, mis pakub taolist teenust.

Nii nagu kirjeldatav lahendus, proovib SEB oma lahendusega säästa loodust. Nende andmed näitavad, et tellerid prindivad umbes 400 000 paberit kuus [9]. Selle kõrval on

välja toodud, et Paberivaba teeninduse kasutamine muudab teenindusprotsessi oluliselt mugavamaks ja minimiseerib arhiveerimisega seotud rahalise kulu.

2013. aastal hakkas tööle Registri ja Infosüsteemide keskuse (RIK) arendatud Delta dokumendihaldussüsteem. Delta on loodud organisatsiooni dokumendihalduse töövahendiks, milles on arendatud dokumendi elukäigu jaoks vajalik funktsionaalsus. Süsteem oskab luua dokumente, otsida neid metaandmete järgi või täistekstiotsinguga ning tänu oma liidestusele DigiSign süsteemiga, neid dokumente ka allkirjastada. Delta süsteem võiks hästi sobida liidestamiseks Paberivaba süsteemiga, kus kliendid ja töötajad saavad dokumente allkirjastada ja hallata [10] [11] [12, 2, 13] [14].

Diplomitöös kirjeldatav lahendus on väga sarnane SEB omale. Kuna diplomitöö autor on olnud SEB töötaja ning on Paberivaba teeninduse arendaja ja hooldaja, siis idee on inspireeritud SEB protsessist, mõttest ja eesmärkidest. Erinevalt SEB pangast, kus lahendus on mõeldud ainult pangasiseseks kasutamiseks, on bakalaureusetöö eesmärk näidata, et on võimalik luua programm ja protsess, mis võimaldab lihtsalt ja mõne konfiguratsiooni muudatusega kasutada seda ka teistes pankades.

2.5 Võimalikud kasutusriigid (turg)

Lahendust on võimalik kasutada riikides, kus on olemas või plaanitakse luua digitaalseid allkirju reguleerivat seadust. Näiteks on lahendus juba teostatav Lätis ja Leedus.

Läti võttis 20. novembril 2002 kasutusele elektroonilise dokumendi seaduse [15]. Elektrooniliste dokumentide loomiseks on Lätis kasutusel eParaksts mobiilne aplikatsioon, ID-kaart füüsilisele isikule ning eraldi ID-kaart juriidilisele isikule [16]. Alates 2017. aasta märtsist pakub Läti koostöös SK ID Solutions AS-iga ka Smart-ID autentimise ja digiallkirjastamise meetodit [17].

Leedu võttis 2018. aasta aprillis kasutusele elektroonilise identiteedi ja usaldusteenused elektrooniliste tehingute kohta (*Electronic Identification and Trust Services for Electronic Transactions*) [18]. Kõige populaarsemad meetodid on loodud koostöös SK ID Solutsionsi ehk endise Sertifitseerimiskeskusega: Mobiil-ID ja Smart-ID. Mobiil-ID-d hakati Leedus pakkuma 2016. ning Smart-ID teenust 2017. aastal [19].

Elektrooniliste dokumentide seadus on kasutusele võetud ka teistes riikides. Näiteks 2009. aastal võttis Soome kasutusele „Tugeva elektroonilise autentimise ja elektroonilise allkirja seaduse“ (*Strong Electronic Identification and Electronic Signatures*) [20], Poolas eksisteerib „Elektroonilise allkirja seadus“ (*Electronic signature*) alates 2001. aasta septembrist.

Vastavate regulatsioonide olemasolul saab mõelda sellele, kuidas kasutada elektroonilise allkirja võimalusi. Paberivaba teeninduse kasutuselevõttu saab kaaluda juhul, kui pangas eksisteerib teenus, mis võimaldab anda digitaalset allkirja.

3 Paberivaba klienditeeninduse lahenduse kirjeldus

3.1 Lahenduse funktsionaalsed nõuded

Lahenduse nõuded katavad ainult skoobis kirjeldatava osa. Kõik nõuded olid valmistatud koostöös SEB Panga tooteomanike ja arendusjuhtidega ning arvestades koostööpartnerite vajadusi ja soove.

Paberivaba klienditeeninduse kõige olulisem aspekt on lihtsus ja arusaadavus kliendi jaoks. Kliendile nähtav protsess peab olema lihtsa ja loogilise ülesehitusega ning lihtsasti jälgitav. Tuleb vältida liigse info näitamist kliendi ekraanil, samas peab kliendile oluline info olema nähtav. Klient peab tundma, et ta kontrollib olukorda ja tegevused, mida temalt oodatakse, on arusaadavad.

Kui klienditeenindaja parasjagu ei teeninda klienti või kui kliendiga ei tehta paberivaba teenindust, siis programm peab olema võimeline kuvama reklaami. Reklaami jaoks on videofail, mida peab olema võimalik vahetada.

Teenindusprotsessi alustamine peab olema sujuv. Tervitus ja klienditeenindaja nime kuvamine ekraanil näitavad viisakust ja austust kliendi vastu. Iga klient peab olema eelnevalt autenditud kasutades digitaalseid autentimisvahendeid. Kliendil peab olema valik, millist vahendit ta tahab kasutada. Eestis peaks valikus olema kõik levinud vahendid – Eesti ID kaart, Smart-ID ja Mobiil-ID.

Paberivaba teeninduse kasutamise ajal kuvatakse kliendi tahvelarvuti ekraanile lepingu/tehingu infot, mida näeb ka töötaja enda arvutiekraanil. Tuleb arvestada, et ainult pangasiseseks kasutamiseks mõeldud andmed peavad olema kliendi ekraanil peidetud. Ekraanil oleva info ja kirja suurus ning väljanägemine peab olema lihtsasti loetav ning arusaadav. Pärast lepingu/tehingu andmete sisestamist allkirjastavad klient ja töötaja tehingu digitaalselt. Selline viis on alati eelistatav lepingute allkirjastamiseks, sest see on kliendile mugavam ja lihtsam, samuti säästab see kliendi aega ning paberikulu. Kõik need aspektid on positiivse kliendikogemuse loomise aluseks. Kliendile väljatöötatud allkirjastamise protsess on internetipangas lihtsasti ligipääsetav ja turvaline. Klient ja pank saavad alati jälgida lepingu staatust. Kõikide digitaalselt allkirjastatud lepingute hoidmine ühes kliendile lihtsalt ligipääsetavas kohas loob kliendile tunde panga

professionaalsusest, turvalisusest ja lihtsustab tema kontrolli oma dokumentide üle. Juhul, kui lepingu allkirjastamine ei õnnestunud, tuleb kliendile sellest teada anda ning anda võimalus allkirjastada uuesti.

Kontakt tuleb lõpetada korrektselt, et ka klient mõistaks, et teenindus on lõppenud. Olulisel kohal on positiivne sõnum või foto, mis tekitab meeldiva emotsiooni. Sõnumeid võiks olla mitu, et korduvalt panka külastavatele klientidele kuvataks erinevaid sõnumeid.

3.2 Lahenduse mittefunktsionaalsed nõuded

Peale selle, et lahendus peab vastama funktsionaalsetele nõuetele, peab süsteem vastama ka mittefunktsionaalsetele nõuetele. Mittefunktsionaalsed nõuded keskenduvad süsteemi käitumise täpsustamisele, mitte konkreetsete funktsionaalsuste või omaduste kirjeldamisele. Sellistele nõuetele tuleb tähelepanu pöörata, et oleks võimalik hinnata süsteemi toimimise kvaliteeti, mitte lihtsalt tegevusi, mida süsteem võimaldab [21].

Paberivaba süsteem peab olema võimeline töötama nii Linuxi serveritel kui ka pilveteenustel nagu Azure, Amazon AWS jm. Kliendirakendus peab töötama ja välja nägema viisakas igas modernses veebilehitsejas. Igas kliendiarvutis peab olema võrguühendus ning loodud vastavad tulemüürireeglid.

Oluline on ka andmete turvalisus. Esiteks peab kogu andmevahetus lahenduse komponentide vahel toimuma SSL ühenduse kaudu. See on turvaprotokoll, mille korral on kogu andmevahetus krüpteeritud. Süsteem ei tohi salvestada ega muul moel säilitada kliendiandmeid. Klientidele peab olema nähtav ainult temaga seotud informatsioon ja lepingud. Selleks, et alustada andmete kuvamist kliendi arvutile, peab isik olema digitaalselt autenditud.

Lahenduse hooldus ei tohi võtta rohkem kui 5% tööajast. See teeb umbes kaks tundi nädalas, kui arvestada 40-tunnise töönädalaga. Seejuures tuleb tagada, et süsteem on saadaval vähemalt 98% pangakontori lahtiolekuajast. Võib eeldada, et kui pangakontorid on suletud, siis võib sulgeda paberivaba lahenduse selle hoolduseks. Tuleb tagada, et kliendid saavad ennast autentida 98% kordadest ehk 100-st autentimisest ainult kahel võiks esineda tõrked, mille puhul peab klient isikutuvastust uuesti proovima.

3.3 Tehnoloogia valik

Arenduskeelte ja tehnoloogia üldine valik on palju mõjutatud sellest, millised tehnoloogiad on pangas kasutusel. Analüüsides seitsme suurema panga töökuulutusi selgus, et kõige populaarsem serverirakenduse arenduskeel on Java. Koguni 37,2% kuulutustest otsitakse just Java arendajaid [22]. Tänapäeval on just Java kõige populaarsem esmane serverirakenduse programmeerimiskeel [23]. See tähendab, et on lihtsam leida inimesi, kes hakkavad rakendust toetama ja edasi arendama. Kahjuks tuleb tõdeda, et mõneti on programmeerimiskeele valik mõjutatud vaadeldavas pangas kasutatavatest keeltest, siis serverirakenduse keeleks tuleb valida Java.

Eessüsteemi (*front-end*) ehituses on tänapäeval kõige populaarsemaks keeleks kujunenud Javascript [24]. Peaaegu kõik SPA lahendused on kirjutatud Javascriptis [25]. Seetõttu kuulub vaatluse alla just Javascript kui kasutajaliidese arenduskeel.

3.3.1 Serverirakenduse arenduskeelte alternatiivid

Serverirakenduse keele valikul on olemas mitu piirangut. Esiteks peab keel olema piisavalt populaarne, et oleks võimalik leida professionaale, kes oskavad arenduskeelt kasutada ja teavad selle detaile. See on oluline kriteerium, sest eeldatakse, et on piisavalt vabu spetsialiste lahenduse edasi arendamiseks või hooldamiseks.

Siinkohal võib vaadata ka ülikoolide poole. Ülikoolis õpetatavad keeled on piisavalt populaarsed ja võimekad, et nendega erinevaid lahendusi luua. Vastavalt Haridus- ja Teadusministeeriumi statistikale lõpetas 2018/2019 õppeaastal info- ja kommunikatsioonitehnoloogia valdkonnas bakalaureuseõppe 394 inimest [26]. See tähendab, et aastas lisandub umbes 400 infotehnoloogia haridusega inimest, kes on valmis oma teadmisi rakendama erinevate programmide loomiseks. Suur eelis on kasutada just ülikoolis õpetatavaid programmeerimiskeeli, sest see ei anna ainult võimalust leida programmeerijaid, vaid ka aidata noortel spetsialistidel alustada oma karjääri.

Keele valikul eeldatakse, et sellel on valikus ka populaarne, funktsionaalne ning erinevate seltside poolt toetatav raamistik. Raamistiku ideed ja vajadust selgitan lähemalt selle bakalaureusetöö raamistiku valiku peatükis.

Vaatame lähemalt, millised oleksid võimalikud Java keele alternatiivid. Valituks osutusid Python, C# ja PHP keele, mis on Eesti ülikoolides IT arenduse programmi sees.

- Python on väga hea objektorienteeritud, interpreteeritav ja interaktiivne programmeerimiskeel, mida võrreldakse sageli selliste keeltega nagu List, Ruby, C# ja Java. Pythonis on kombineeritud märkimisväärne võimekus ja lihtne süntaks. Keel pakub nii mooduleid, klasse, pakkimist kui ka dünaamilisi andmetüüpe [27]. Keele valikus on ka palju raamistike. Kõige populaarsem on Django raamistik [28]. Django on kõrgetasemeline Pythoni veebiraamistik, mis soodustab kiiret arendamist ja puhast programmi struktuuri. Django võtab enda peale palju veebiarendusega seotud vaevasid, et programmeerija saaks keskenduda just programmi ehitamisele ja ei peaks juba olemasolevat lahendust uuesti looma [29].

- C# on lihtne ja kaasaegne objektorienteeritud ning tüübikindel programmeerimiskeel. C# keel pärineb C-keelte perekonnast ja on tuttav C, C++ ja Java arendajatele. Keel on loonud Microsoft [30]. C# keelt kasutatakse nii veebiarenduses kui ka Windowsi aplikatsioonide loomisel ja isegi mängude arenduses. Viimase kohta võib öelda, et Unity on üks tuntumaid mängumootoreid ja C# oskab väga hästi selle mootoriga integreeruda. Tänu lihtsusele ja hõlpsasti kasutatavatele funktsioonidele pakub C# algajatele üsna madalat õppekõverat [31]. Kõige populaarsem keele raamistik on ASP.NET Core [32]. Raamistik on loodud selleks, et võimaldada käitamise (*runtime*) komponentidel, API kompilaatoritel ja keeltel kiiresti areneda, pakkudes siiski rakenduse töötamiseks stabiilset platvormi [33].

- PHP on laialt kasutatav avatud lähtekoodiga üldotstarbeline skriptikeel, mis sobib eriti hästi veebirakenduse loomiseks ja mida saame HTML-i sisse põimida. Erinevalt Javascriptist käivitatakse PHP kood serveris, mis genereerib HTMLi lehe ja mis saadetakse kliendile. Klient saab kätte ainult tulemuse ning ei saa teada, milline kood sellise tulemuse saamiseks jooksis. Keel on äärmiselt lihtne uutele arendajatele, aga pakub palju funktsioone ka professionaalsetele arendajatele [34]. Kõige populaarsem PHP raamistik on Laravel. Laravel loodi 2011. aastal ning on saanud kõige populaarsemaks vabavaraliseks PHP raamistikuks maailmas. See lihtsustab arendusprotsessi, lihtsustades tavapäraseid toiminguid nagu marsruutimine, salvestamine vahemällu, autentimine jpm [35].

Analüüsis selgus, et Java keelele on palju alternatiive. Vaatamata suurele valikule otsustatakse valida just Java keel, kuna keele valik oli mõjutatud vaadeldava panga soovidest.

3.3.2 Serverirakenduse raamistiku valimine

Varasema analüüsi käigus oli otsustatud, et serverirakenduse programmeerimiskeele valikuks osutus Java keel. Tänapäeval on levinud ja mugav kasutada erinevaid raamistike. Raamistik on mingi kindla tegevuse lihtsustamiseks mõeldud abivahend või vahendite kogum [36]. Lisaks abiteekidele sisaldab programmi arhitektuuriliste probleemide lahendamiseks või vältimiseks mõeldud meetodeid [36].

Raamistikud võimaldavad projektiga kiiresti alustada. Nad võimaldavad koodi taaskasutamist ning mitmel osapoolel panustada programmi loomisesse ja hooldamisse. Raamistiku kasutamine peab aitama efektiivsemalt arendada [37].

Seega raamistike kasutus on tähtis. Järgmiselt vaadeldakse erinevaid võimalikke Java raamistike:

- Spring – teiste Java raamistike seast absoluutne liider. Springi raamistiku teadmine on üks levinumaid nõudeid Java arendaja jaoks. See on võimas ja kerge Java raamistik. Programmeerijad ise ütlevad, et Spring teeb Javast lihtsama, kaasaegsema, produktiivsema ja pilveteenusteks valmis keele. See on tuntud oma sõltuvuse süstimise ja aspektidele suunatud programmeerimisfunktsioonide poolest. Raamistiku plussideks on modulaarsuse toetus, lihtne programmi testimine, väga suur ökosüsteem, laialdane dokumentatsioon ja palju materjali õppimiseks. Miinusteks on väga järsk õppimiskõver ning raske raamistiku seadistus [38].

- Spring Boot – see on osa Java raamistikust Spring, mis on disainitud võimaldamaks luua kvaliteetset Springi rakendust kiirelt ja lihtsalt. Spring Boot valib rakenduse seadistused ja kolmanda osapoole teeke arendaja eest, mis võimaldab rakenduse loomisel keskenduda arendamisele ja mitte seadistamisele. Enamus rakendusi vajab vähe seadistamist, kuid Spring Boot jätab võimaluse seadistada rakendust samaväärselt Spring -ga [39].

- Play – veel üks väga kerge raamistik, mida paljud arendajad naudivad. See võimaldab ehitada veebirakendusi kasutades Java või Scala arenduskeelt. Alguses oli raamistik mõeldud rahuldamaks kaasaegsete mobiili- ja veebirakenduste nõudmisi. Play põhineb veebisõbralikul ja kergekaalulisel arhitektuuril. Erinevused teistest raamistikest on kiirus, kvaliteet ja valmidus mastabeerimiseks. Raamistikule ehitatud koodi on lihtne siluda ja konfigureerida. Ta tõstab arendajate efektiivsust, kuna toetab kuumale koodi uuendamisele (*hot reload*). Miinuseks võib märkida, et raamistik on asünkroonne, mis alati ei sobi ning mille kirjutamise põhimõtete õppimine võib olla keeruline [38].

Java maailmas on ka palju muid raamistike. Selles töös olid vaatlusel ainult need kolm. Arvestades suurt populaarsust, paindlikku konfigureerimist, suurt hulka dokumentatsiooni ja õppimismaterjali, sai uuringu käigus Paberivaba lahenduse ehitamiseks valitud Spring Booti raamistik. Vaatamata eeltoodud plussidele leidub ka palju inimesi, kes oskavad seda raamistiku kasutada. Märkimisväärne on ka see, et Spring Boot on ülikoolides Java õppeainete osa.

3.3.3 Kasutajaliidese raamistiku valimine

Varasemalt toodi välja, et Paberivaba lahenduse arendamiseks võetakse kasutusse JavaScripti programmeerimiskeel. Keel on iseenesest väga võimekas, aga selleks, et ehitada valmis SPA kliendi aplikatsioon, oleks otstarbekas valida raamistik, mida kasutada kliendirakenduse loomiseks.

Tänapäeval on kõige populaarsemad SPA raamistikud Angular, React ja Vue.js [40]. Need raamistikud ei ole ainsad, ehkki nendest räägitakse kõige rohkem. Võib välja tuua ka selliseid raamistike nagu Ember.js, Mithril, Aurelia ja Backbone.js [41].

Vaatleme lähemalt neist kolme kõige populaarsemat:

- Angular – üks võimsamaid, tõhusamaid ja avatud lähtekoodiga JavaScripti raamistik. Loodud Google'i poolt 2010. aastal, kuid 2016. aastal kirjutati see täiesti ümber, luues uue raamistiku Angular 2+. See on kõikehõlmav lahendus ja raamistik, mitte erinevate teekide kogum. Arendajad saavad rohkem keskenduda koodi kirjutamisele, mitte nende nõudmistele vastavate teekide otsimisele. See on omadus, mis teeb Angulari raamistiku saadaolevatest üheks parimaks. Raamistik

on loodud TypeScripti keele põhjal, mis toob kaasa kõik selle pakutavad eelised: noolefunktsioonid, asünkroonsus, klasside süntaks. Raamistik on loodud suurtes meeskondades töötamiseks nii, et iga meeskonnaliige saab töötada oma koodiosaga kartmata, et ta lõhub midagi kellelgi teisel. Angulari plussideks võib pidada komponendipõhist arhitektuuri, TypeScripti kui põhikeele kasutust, suurt jõudlust, suurt ökosüsteemi ja Material Design'i teekide kasutust. Nõrkadeks kohtadeks on aga raamistiku keerukus, õppimise raskused ja mõnede komponentide halb dokumentatsioon. [41] [40]

- React on Facebooki ja Instagrami arendatud avatud lähtekoodiga teek eesmärgiga kergendada interaktiivsete, taaskasutatavate kasutajaliidese komponentide loomist. Reacti on võimalik teiste teekide ja raamistikega koos kasutada. Reacti arendajad on loonud raamistiku eesmärgiga lahendada probleemi, mis on seotud suurte rakenduste ehitamiseks, kus andmed muutuvad pidevalt [42]. Reacti raamistikul on palju eeliseid ja vähesel määral negatiivseid aspekte. Plussideks on näiteks virtuaalne DOM, mis parandab nii kasutajakogemust kui arendajate tööd. Virtuaalne DOM aitab isoleeritud komponentide rakendamisega värskendada mis tahes kasutaja muudatusi ilma teiste osade sekkumiseta. React -le ehitatud kood on stabiilne, kuna toetab ühesuunalist andmevoogu. Kogu andmete sidumine toimub seega ülevalt alla. See on avatud lähtekoodi ja erinevate tööriistadega teek ning igasugused muudatused ja uuendused jõuavad kiiresti kogukonnani. Miinustena võib aga välja tuua, et raamistiku õppimiskõver on suhteliselt pikk. Reacti kõikide detailide õppimine võtab rohkem aega kui näiteks Angulari õppimisel. Seoses sagedastele muudatustele raamistikus on mõnikord raske leida uuendatud dokumentatsiooni [40].

- Vue.js – ehkki see JavaScripti raamistik ehitati alles 2016. aastal, on see juba turule jõudnud ja tõestanud oma väärtust erinevate funktsioonide pakkumisega. Väga paljud saadavalolevad funktsioonid meelitavad arendajaid seda raamistiku kasutama. Raamistikul on loogiline struktuur ja failide paigutus. Ta oskab kahepoolset reaktiivset andmesidumist ning ei nõua selleks lisateeke, säilitades oma paindlikkuse. Vue.js on võrreldes teiste SPA raamistikega tõeliselt väike, mis tõestab selle võimekust ja kiirust teiste seast. Vue.js on tuntud oma kõikehõlmava dokumentatsiooni poolest, mis säästab arendajate aega ning kiirendab raamistiku

õppimist. Selle plussideks on raamistiku selgus ja lihtsus, lihtne integreeritavus ja koodi taaskasutus. Miinusteks on välja toodud liiga suur paindlikkus, mis toob kaasa mõnede eeskirjade eiramise. Samuti on välja toodud, et Vue.js kogukond ei ole veel väga suur. Muudatusi ja parandusi viivad sisse pigem üksikud arendajad. Samas tuleb märkida, et kasutajaskond suureneb iga päevaga [40] [41].

Analüüsides võimalikke SPA lahenduste raamistike, osutus väljavalituks Angular. Angular on nii arendajate kui ka selle kogukonna seas populaarne. See tähendab, et arendajate leidmine ja värbamine on lihtsam. Reacti kogukond on samuti suur, kuid see raamistik ei kata teisi nõudmisi.

Angular on mõeldud suurtele programmidele ja korporatsioonidele. Kuna Paberivaba lahendus on pigem mõeldud pankadele, võib pidada kirjeldatavat lahendust suureks süsteemiks. See eeldab, et lahendusega hakkab töötama mitu arendajat. Angular on hea, sest võimaldab mitmel programmeerijal tööd teha üksteise tööd häirimata.

Oluline on ka see, et Angular pakub kohe kõiki vajalike komponente ja teeke, et koodi saaks hakata kiiresti kirjutama. See tähendab, et arendajad ei pea otsima juurde näiteks sessioonihalduseks, autentimiseks või marsruutimiseks mõeldud teeke. Kõik on olemas Angulari raamistikus. Suur eelis on ka TypeScripti keele kasutus, mis võimaldab luua tüüpkindlaid muutujaid ning seega vähendada koodis tekkivaid vigu.

3.3.4 Andmebaasi valimine

Paberivaba lahenduse töötamiseks on vajalik andmebaas. Eelkõige on see vajalik selleks, et hoida Paberivaba lahenduse konfiguratsioone, reklaamfaile ja et oleks koht, kus on omavahel seotud klienditeenindaja ja kliendi arvuti. Täpsemalt sellest, milleks on see vajalik, kirjeldan lahenduse arhitektuuri osas.

Andmebaas on organiseeritud andmete hulk arvutis, kuhu andmebaasi programmide abil saab andmeid lisada, parandada ja eemaldada ja millest saab teha mitmesuguseid aruandeid. Lisaks tuleb märkida, et andmebaas hoiab omavahel seotud informatsiooni ning hoitavad andmed on kirjeldatud andmekirjeldusega, mida hoitakse koos andmetega. Vahendid, mida kasutatakse andmete kirjeldamiseks, määravad andmebaasi mudeli [43].

Tänapäeval leidub palju erinevaid andmebaase. Võib eristada kahte tüüpi – relatsioonilised ja mitteseotud andmebaasimudelid. Relatsioonilise andmebaasimudeli

korral salvestatakse andmed tabelitesse. Tabel koosneb ridadest ja veergudest, kus read esindavad igat üksust, samas kui veerud esindavad atribuute. Relatsioonilistes andmebaasides teostatakse seoseid erinevate tabelite vahel andmeväljade väärtuste kaudu [44] [45].

Relatsiooniline andmebaas ei ole efektiivne suure hulga andmete, näiteks suurandmete salvestamiseks. Mitteseotud andmebaas on selle probleemi lahendus. Lisaks nimetatakse ka mitteseotud andmebaasi NoSQL andmebaasideks. Need andmebaasid võivad salvestada suuri andmeid. Olemas on mitu erinevat mitteseotud andmebaasi tüüpi. Sellisteks on näiteks dokumendi andmebaas, mida kasutatakse dünaamiliste andmete salvestamiseks ning kus salvestatavad andmed on JSON formaadis. Välja võib tuua ka vahemälu andmebaasid, kus andmed salvestatakse kettale või vahemällu [44].

Töös püstitatud eesmärgi saavutamiseks piisab relatsioonilistest andmebaasides, sest andmebaasis hoitavate andmete maht on väga väike. Lisaks on tarvis luua seoseid erinevate tabelite vahel. Valiku kriteeriumiks on kiire installeeritavus, lihtne hooldus, Spring Booti raamistiku toetamine. Järgnevalt vaadeldakse erinevaid valikuid, mis vastavad antud kriteeriumitele:

- PostgreSQL on objekt-relatsiooniline andmebaasi juhtimissüsteem. PostgreSQL on tasuta ja avatud lähtekoodiga tarkvara. Võrreldes paljude teiste avatud lähtekoodiga programmidega ei kontrolli PostgreSQL'i ükski konkreetne ettevõte. Programmi arendusega tegeleb globaalne kogukond, mis koosneb erinevatest arendajatest ja firmadest. PostgreSQL oli üks esimesi andmebaase, mille funktsionaalsuse hulka kuulus multiversioon konkurentsjuhtimine - meetod ühiskasutusega andmebaaside jõudluse tõstmiseks, et loobuda rea- ja tabelilukustusest süsteemi efektiivsuse parandamiseks. PostgreSQL'i plussidena võib välja tuua lihtsat tuge, pärimist, võimsaid ja usaldusväärseid tehingute ja replikatsioonide mehhanisme ning ta toetab praktiliselt piiramatut andmebaasi suurust [46]. Miinusteks on välja toodud, et on kordi, kus andmebaasi taastamisel tekivad varaandmetest mitmesugused väljakutsed, millega tuleb tegeleda [13].

- Oracle Database on Oracle Corporationi arendatud ja turustatud mitmemudeliline andmebaaside haldussüsteem. 2020. aasta seisuga on kõige uuem Oracle'i andmebaasi versioon Oracle Database 19c. Erinevalt PostgreSQL'i andmebaasist

ei ole Oracle andmebaas vabavaraline ja selle kasutamine maksab. 19c versioonis on väga palju tähelepanu pööratud just sellele, et andmebaas oleks saadaval kui pilveteenus. Samuti toetab andmebaas JSON formaati ning võimalust andmebaasi killustada (*sharding*) [47].

- MySQL on üks mitmest relatsioonandmebaasi haldussüsteemist, mis pakub mitut eelist. Andmebaasi haldussüsteem aitab muuta andmed turvaliseks ja säilitada andmete terviklikkuse. Lisaks saavad kasutajad teha varukoopiaid. Tavaliselt kasutatakse MySQL'i andmebaasi koos PHP arenduskeelega [48]. Andmebaasi plussiks on see, et see on tasuta kasutatav. Võib kasutada ka vabavaralist MariaDB -d, mis on lihtsalt vabavaraline versioon MySQL'i andmebaasist. Antud haldussüsteemi on väga lihtne paigaldada ning see ei nõua palju arvutimälu või protsessori võimsust. Selle taga seisab suur kogukond, mis võimaldab dokumentatsiooni kättesaadavust. Miinusena võib välja tuua, et andmebaasi saab paigaldada ainult serverile. Paigaldamine kaasakantavatesse seadmetesse ei ole lubatud [49].

Peale erinevate haldussüsteemide analüüsi võib järeldada, et Paberivaba lahenduse jaoks sobib kõige paremini PostgreSQL andmebaas. Seda on lihtne paigaldada, ta on vabavaraline, relatsioonilise mudeliga ning selle taga seisab suur kogukond. Erinevalt MySQL'i andmebaasist, ei oma ta suuri piiranguid andmemahtudes ja võimaldab kiiresti ja vaevata luua replikatsioone, mis on tähtis, kui lahendust hakatakse jooksutama pilves.

3.3.5 Versioonihalduskeskkonna valik

Versioonihaldus on vajalik selleks, et võimaldada arendajatel oma kirjutatud koodi versioneerida. Esiteks võimaldab süsteemide kasutamine koodi paremat haldust. Näiteks on võimalik näha, millised muudatused ning kes ja millal on teinud, miks need on tehtud ning vajadusel saab koodi taastada. Eriti kasulik on see siis, kui ühe projektiga tegeleb mitu programmeerijat [50].

Versioonihalduskeskkonnad põhinevad peamiselt Git funktsionaalsusel ning on loodud selleks, et neid funktsionaalsusi laiendada ja mugavamaks teha. Peale Git versioonihalduse on olemas ka teisi, näiteks SVN, Mercurial ja paljud teised [50]. Tänapäeval on Gitil põhinevad kõige populaarsemad versioonihalduskeskkonnad GitHub, GitLab ja Bitbucket [51]. Järgnevalt vaadeldakse neid lähemalt:

- GitHub loodi 2007. aastal GitHub Inc. korporatsiooni poolt San Franciscos [52]. GitHub on oma iseloomult Git repositooriumite hostimise teenus. Iga soovija saab tasuta avada konto ning hallata oma repositooriumit. Tuleb märkida, et GitHub on saanud väga populaarseks kohaks, kuhu paigutatakse avatud lähtekoodiga projekte [53]. 2019. aasta seisuga hostib GitHub umbes 100 miljonit repositooriumit [52].

- GitLab on teine Git repositooriumite hostimise teenus, mis loodi GitLab Inci poolt. GitLabi kohta öeldakse, et see on täielik DevOpsi platvorm, mis on saadaval ühe rakendusena. GitLab on vabavaraline ja erinevalt GitHubi versioonihalduskeskkonnast, on seda võimalik paigaldada ka ükskõik millistele serveritele, sh nendele, mida firma hostib ise. See võimaldab luua täieliku kontrolli selle üle, mis inimesed milliseid ressursse saavad kasutada ja kuidas on kõik GitLabis hallatud [51] [54].

- BitBucket on Git repositooriumite haldamise süsteem, mis annab meeskondadele ühe koha oma projektide planeerimiseks, koodiga töötamiseks, testimiseks ja juurutamiseks. Selles versioonihalduskeskkonnas on võimalik luua tasuta piiramatu arv repositooriume. BitBucket on loodud ettevõtte Atlassian poolt, kelle üheks tooteks on ka Jira [55]. Sellepärast on BitBucketi versioonihaldussüsteemil ka integratsioon Jiraga. Lisaks on BitBucketi sisse ehitatud CI/CD, mis võimaldab koodi testida ja juurutada serveritele [56].

Analüüsides valikuid, jõuti järeldusele, et Paberivaba projekti jaoks sobib kõige paremini BitBucketi versioonihaldussüsteem. Süsteemi eelis on integratsioon Jiraga, mis on laialdaselt kasutatav pankades [57]. Samuti sisse ehitatud CI/CD võimaldab kiiresti ja mugavalt projekti juurutada.

3.3.6 Arenduskeskkonna valik

Java programmeerimiskeelele mõeldud arenduskeskkondi leidub palju. Arvatavasti kõige tuntumad ja populaarseimad nendest on Eclipse ja IntelliJ IDEA [58]. Viimasest eraldi leidub ka IntelliJ IDEA Community Edition versioon, mis on erinevalt oma suuremast vennast arendajatele täiesti tasuta saadaval. Järgnevalt vaadeldakse neid arenduskeskkondi lähemalt:

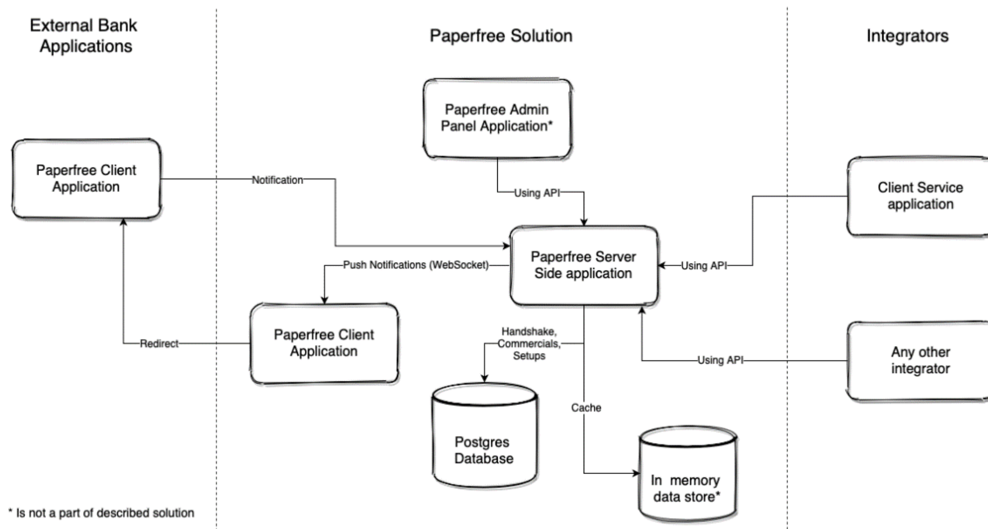
- Eclipse on integreeritud arenduskeskkond, mille töötas välja IBM, asutades Eclipse Foundationi 2004. aastal. Eclipse on vabavaraline tarkvara, mis on peamiselt kirjutatud Java keeles ja seda kasutatakse samuti peamiselt Java rakenduste arendamisel. Erinevate pistikprogrammide (*plugin*) kasutamisel on aga võimalik Eclipse'i funktsionaalsust laiendada, mis võimaldab arenduskeskkonda kasutada ka teistes keeltes kirjutatavate rakenduste arendamisel [59].

- IntelliJ IDEA on integreeritud arenduskeskkond, mis on suures osas mõeldud Java rakenduste loomisele. Keskkonna on välja töötanud ettevõtte nimega JetBrains, mis varasemalt kandis nime IntelliJ. Esimene versioon sellest sai saadavaks 2001. aastal. IntelliJ IDEA on saadaval kahes erinevas versioonis – Community Edition ja Ultimate Edition. Esimene on tasuta saadaval ja on litsentseeritud Apache 2.0 litsentsi all. Teine on kommertsversioon. Mõlemat saab kasutada müüdavate lahenduste arendamiseks. IntelliJ IDEA erineb teistest oma kasutusmugavuse, paindlikkuse ja kindla disaini poolest, mis on kasutusel kõikides JetBrainsi toodetes. Suurim põhjus, miks IntelliJ IDEA-d peetakse üheks parimaks Java põhiste programmilahenduste loomise keskkonnaks, on selle abifunktsioonid, mis muudavad keskkonna kasutust lihtsalt kasutatavaks. Samuti leidub seal täiustatud vigade kontrollimise süsteem, mis võimaldab kiiremat ja lihtsamat võimalike koodivigade tuvastamist [60].

Arvestades autori kogemust ja pikemat kokkupuudet JetBrainsi toodetega, tehti valik IntelliJ IDEA Community Editioni kasuks.

3.4 Lahenduse arhitektuur

Paberivaba lahendus koosneb kolmest erinevast programmist ja andmebaasist. Selles töös on vaatlusel kaks programmi: serverirakenduse programm, mis on ehitatud Java peal ja kasutab Spring Booti raamistikku ning kliendi rakendus, mis on ehitatud Typescripti ja Angulari raamistikul. Kolmas aplikatsioon, mis on töötajate vaade ning mõeldud just lahenduse konfigureerimiseks jääb selle töö skoobist välja.



Joonis 1. Paberivaba lahenduse arhitektuur (autori looming).

Arhitektuuriline pilt on jaotatud kolme kategooriasse – rakendused, mis tahavad integreeruda Paberivaba lahendusega, programmid, mida lahendus kasutab ning Paberivaba lahenduse enda komponendid.

Esiteks on Paberivaba lahenduse oma komponendid. Komponentideks on serveri rakendus, mis representeerib Paberivaba API. Teiseks on kliendi rakendus, mis on nähtav kliendi arvutis ning mis on otse seotud paberivaba APIga. Samasse tulpa kuulub ka Postgresi andmebaas, mille eesmärk on hoida lahendusega seotud konfiguratsioone ning vaadete malle.

Välja on toodud rakendused, mis hakkavad integreeruma paberivaba programmiga. See võib olla näiteks programm, millega töötavad kliendinõustajad. Juhul, kui see programm on konfigureeritud või arendatud nii, et nõustaja vaade peab olema jagatud ka kliendiga, saadab nõustaja aplikatsioon REST päringu paberivaba API vastu. Selleks on loodud kirjeldatava lahenduse API poolel REST punktid, kus võetakse vastu andmed ja käsk, mida tuleb paberivabal lahendusel teha. See võib olla näiteks kliendile informatsiooni jagamine, dokumendi allkirjastamine või ka sessiooni lõpetamine ja kliendile reklaami kuvamine. Kõik käsud on eelnevalt defineeritud ning telleri programm sisaldab loogikat, millal millist käsku tuleb saata.

Kolmandas tulbas on välja toodud programmid, mida kasutab paberivaba lahendus ise. Praegu selliseks teenuseks on dokumentide allkirjastamise teenus. Nagu eelnevalt

kirjeldatud, on üheks eelduseks allkirjastamise protsessi ning vastava kliendivaate olemasolu pangas. Juhul, kui kliendinõustaja programm saadab paberivabale lahendusele teate, et tahetakse dokumenti allkirjastada, suunab paberivaba lahendus kliendi allkirjastamise lehele, kus kõiki toiminguid saab teostada. Kui dokumendi allkirjastamine on edukas, suunatakse klient tagasi paberivabasse lahendusse, kus talle kuvatakse teavitus edukast allkirjastamisest.

3.4.1 Protsessi kirjeldus

Vaatleme ühte protsessi juhul, kui kliendi arvutis tuleb kuvada andmeid, mida kliendinõustaja oma ekraanil kliendiga töötades näeb.

Enne kui klient saab näha talle kuuluvaid andmeid, tuleb tal ennast identifitseerida. Vajutades kliendi arvuti ekraanile ilmub talle valik kolmest autentimise meetodist – Smart-ID, Mobiil-ID ja ID-kaart. Klient saab ise valida, millist meetodit ta tahab kasutada. Sisestades vajalikud andmed, saadetakse päring autentimisteenusele, mis tagastab teate, kas autentimine oli edukas või mitte. Juhul, kui protsessi käigus ei olnud probleeme, on kliendi arvuti valmis vastu võtma talle kuuluvaid andmeid.

Kui kliendinõustaja läheb oma programmis vaatesse, mille andmed tuleb jagada ka kliendiga, saadab nõustaja programmi REST päringu paberivaba API vastu. API töötleb saadetud päringut ning otsustab, mida ja kuhu tuleb edasi saata. Kuna üks serverirakendus võib töötada mitme kliendirakenduse heaks, tuleb tal kindlasti teada, millisele kliendirakendusele antud andmed kuuluvad. Selle probleemi lahendamiseks kasutatakse kaardistamist. Eelnevalt on andmebaasis seadistatud, missugune kliendinõustaja arvuti on seotud millise kliendi arvutiga. Kasutatakse üks-ühele kaardistamist, kus ühel poolel on kliendinõustaja arvuti nimi ning teisel poolel kliendi arvuti nimi. Seega kui API saab kätte uue päringu, peavad kätte saadud andmed sisaldama ka informatsiooni, millisest arvutist päring saadeti. Serverirakendus pöördub siis andmebaasi poole ning pärib sellist kirjet, kus kliendiarvuti nimi on see, mis kätte saadi. Nii leitakse kliendi arvuti, kuhu andmed tuleb saata.

Kui nõustaja arvutile vastav kliendi arvuti on leitud, tuleb koostada päring ning see edasi saata kliendirakendusele. Saatmiseks kasutatakse WebSocketi tehnoloogiat. WebSocket on selline tehnoloogia, mis võimaldab pidada brauseri ja serveri vahel interaktiivseid kahesuunalisi seansse [61]. Teisisõnu on see tehnoloogia, mis võimaldab serveri

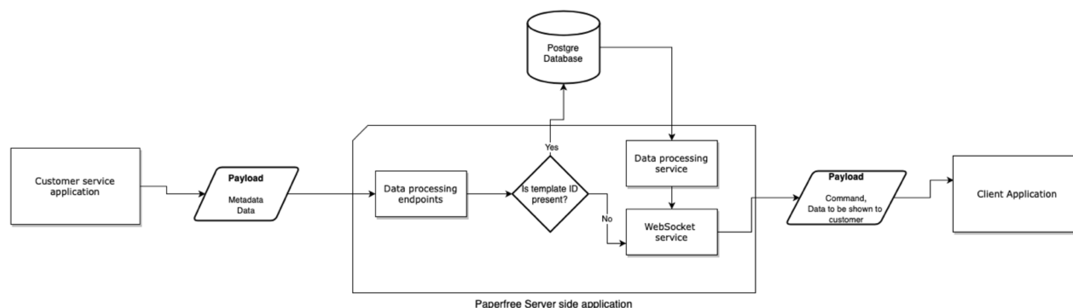
rakendusel saata sõnum kliendi rakendusele otse ja ilma, et kliendirakendus peaks kogu aeg küsima, kas talle on uusi andmeid või mitte. Kliendirakendusel registreeritakse nii öelda kuulaja unikaalse nimega, milleks antud lahenduse puhul on kliendi arvuti nimi. Kuna server teab, mis on kliendi arvuti nimi, siis ta teab ka järjekorda, kuhu päring saata. Nii saab kliendirakendus kätte sellele kuuluvad andmed.

Kui kliendirakendus on oma käsu koos andmetega saanud, käitub rakendus vastavalt saadetud reeglitele. Selleks võib olla näiteks kliendile informatsiooni näitamine, kliendiga seansi lõpetamine ja reklaamide näitamine või siis kliendi suunamine allkirjastamise lehele. Kliendirakendus on eelnevalt programmeeritud nii, et aru saada erinevatest käskudest ja vastavalt sellele ka käituda. Võimalusi, kuidas kliendile infot näidata, analüüsin kliendirakenduse arenduse peatükis.

Sellist arhitektuuri kasutades võib märkida, et süsteemid, mis hakkavad kasutama Paberivaba lahendust ei ole määratud. Ükskõik milline programm, mis tahab kliendile infot näidata, võib APIt kasutades hakata saatma need kliendi arvutile. Vajalik on vaid eelnev kaardistamine ja mallide olemasolu, et lahendus teaks, millises vormingus andmed tuleb kuvada.

3.4.2 Andmete voog

Vaatleme lähemalt, kuidas liiguvad andmed erinevate komponentide vahel. Kõige tähtsam, mida tuleb märkida on see, et paberivaba lahendus ei salvesta ega hoia oma mälus andmeid, mis talle on saadetud. Vaatleme andmete voogu uuesti eelneva stsenaariumi näitel.



Joonis 2. Andmete voog (autori looming).

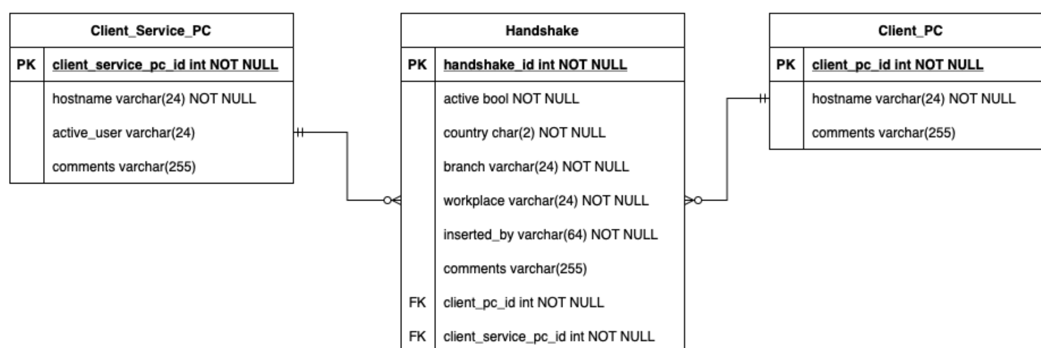
Kliendinõustaja programm on tsentraalne koht, kus kogutakse kokku kõik andmed, mida tahetakse kuvada paberivaba lahenduse kliendirakenduses. Need andmed saadetakse läbi HTTP või HTTPSi protokolli paberivaba APIsse. Andmete kõrval asetsevad ka metaandmed. Nendes kajastub, kes saatis päringu, millisest nõustaja arvutist see tuleb, millised on andmed, mida tuleb kuvada ning logi jaoks vajalikud andmed. Nendeks on unikaalne päringu ID ja aeg, millal sõnum kliendinõustaja programmist välja saadeti.

Saadud andmeid protsessitakse paberivaba serverirakenduses. Kui metaandmetes on öeldud, et kliendivaade tuleb koostada mallist, siis minnakse ja päritakse andmebaasist vastav mall. Peale seda asendatakse mallis olevad kohahoidjad reaalsete andmetega. Vastasel juhul kui metaandmetes puudub informatsioon, et tuleb kasutada malli, saadetakse andmed otse kliendirakendusele.

Kasutades WebSocketi tehnoloogiat, saadetakse (*push*) andmed kliendirakendusele. Koos nendega saadetakse ka käsk Angulari aplikatsioonile. Vaadeldavas näites saadetakse käsk kuvada kliendile andmeid. Peale seda kui kliendirakendus kontrollib, et vastav klient on edukalt autenditud, kuvatakse saadud andmed ekraanile.

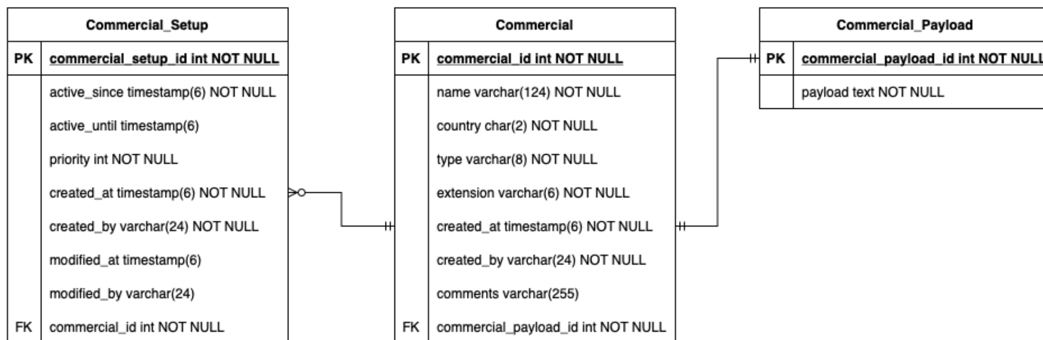
3.4.3 Andmebaasi mudel

Lähtuvalt ärinõuetest ja eeldusest, et paberivaba süsteem ei salvesta andmeid, on joonistel 3 ja 4 välja toodud andmebaasi mudelid, mis katavad erinevat funktsionaalsust. Tabelid ei ole üksteisega seotud, kuna neil pole ei funktsionaalset ega ärilist kokkupuutepunkti.



Joonis 3. Kliendi arvuti ja nõustaja arvuti kaardistuse andmebaasi mudel (autori looming).

Joonisel 3 on välja toodud kliendi arvuti ja kliendinõustaja arvuti kaardistamise olemite vaheline suhe. Selleks, et oleks võimalik hoida ajalugu, on loodud eraldi kliendi arvuti ja nõustaja arvuti olemid, mis on seotud kokku kolmanda olemiga.



Joonis 4. Kliendi arvuti reklaami salvestamise andmebaasi mudel (autori looming).

Joonisel 4 on näidatud andmebaasi mudel, kus hoitakse informatsiooni kuvatavatest reklaamidest. Aluseks on võetud reklaami olem, millele võib vastata mitu erinevat reklaami seadet. See on tingitud ärinõuetest – reklaami peab olema võimalik konfigureerida nii, et oleks võimalik valida, mis perioodil videofaili klientidele näidata. Kui tekib konflikt, et ühel ja samal ajal leidub kaks või enam kirjet, mis vastavad kriteeriumitele, siis valitakse see, millisel on suurem järk. Faili sisu on välja toodud eraldi olemisse. Põhjus on see, et saaks pärida reklaamide andmebaasi kirjeid ilma, et iga kord peaks kaasa küsima kogu faili sisu. Ainult siis, kui reklaam on välja valitud, pöördutakse andmebaasi poole ja küsitakse faili.

3.4.4 Võimalik arhitektuuriline optimeerimine

Serverirakenduse tsentraalseks kohaks on andmete edasi saatmine kliendirakendusse. Selleks on iga päringu korral vaja andmebaasist teada saada, millisele kliendi arvutile vastab päringus saadud nõustaja arvuti.

Selleks, et optimeerida ja kiirendada antud kohta, võib kasutada ka mälupõhist andmebaasi. See on andmebaas, mis talletab andmed muutmälus. Tänu sellele lüheneb ka andmete kättesaamise aeg. Lähteandmed laaditakse mällu üles tihendatult, mitte-relatsioonilises vormingus [62]. Üks võimalikest mälupõhistest ja üks populaarseim nendest on Redis [63] [64].

Paberivaba lahenduses oleks võimalik kasutada mälupõhist andmebaasi talletamaks kaardistamise informatsiooni. Kuna iga päringu korral tuleb leida vaste, siis on tähtis, et see osa töötaks kiiresti ja ei koormaks andmebaasi.

3.5 Analüüsi kokkuvõtte

Analüüsis oli lähemalt vaadeldud paberivaba lahenduse funktsionaalseid ja mittefunktsionaalseid nõudeid. Sellega tahtsin näidata, mis on lahenduse idee ning millele peab lõpptulemus vastama.

Järgmisena analüüsiti tehnoloogilisi võimalusi ja valikuid. Võrreldes mitmeid serverirakenduse keeli ja raamistike ning kliendirakenduse keeli, jõuti järelduseni, et kasutusse läheb Java koos Spring Booti raamistikuga serveri poole peal ning TypeScript koos Angular -ga kliendi poolel. Need valikud olid suuresti mõjutatud ettevõtete enda nõuetest. Vaadati, millised on võimalikud versioonihaldus- ja arenduskeskkonnad. BitBucket oli valitud kui versioonihaldussüsteem tänu selle lihtsale integreerumisele Jiraga. IntelliJ IDEA valik oli aga tingitud autori enda eelistustest.

Analüüsiti lähemalt lahenduse arhitektuuri. Vaadati, millistest osadest antud programm koosneb ning kuidas ning milliste protokollidega on need omavahel seotud. Uuriti lähemalt, kuidas näeb välja andmevoog erinevate komponentide vahel ning milline on andmebaasi mudel. Lõpuks toodi välja ka soovitusi, kuidas saaks optimeerida serverirakenduse tööd, kasutades mälupõhist andmebaasi.

4 Lahenduse arendus

Selles peatükis vaadeldakse lähemalt serverirakenduse ja kliendirakenduse tehnilisi lahendusi. Kolmanda suurema osana näidatakse, kuidas teostati programmiosade testimist.

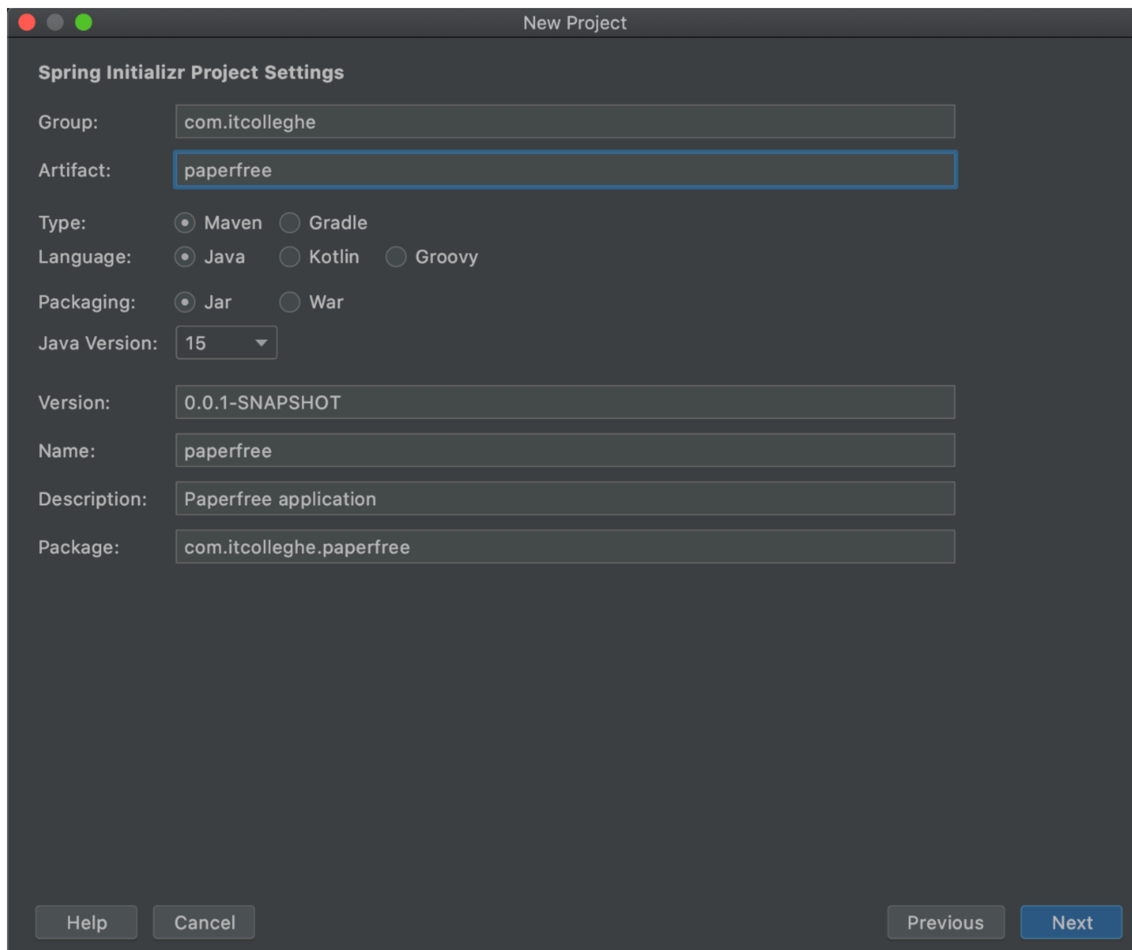
4.1 Serveri rakenduse arendus

Paberivaba lahenduse analüüsi käigus valiti serverirakenduse programmeerimiskeeleks Java koos Spring Boot raamistikuga. Järgnevalt vaadeldakse, kuidas rakendus loodi, milline on selle struktuur, kõige tähtsamad konfiguratsioonid ning REST API kirjeldus. Rakendus vastutab äriloogika eest, omab ühendust andmebaasiga konfiguratsioonide talletamiseks ning on kliendirakenduse ühenduslüli.

Serverrakendus oli loodud jälgides objektorienteeritud programmeerimise põhimõtteid [65]. Samuti rakendati puhta arhitektuuri ehk SOLID printsiipe. Jälgides seda printsiipi on võimalik kirjutada programm, mis on kergemini hallatav ja laiendatav [66].

4.1.1 Spring Boot rakenduse loomine

Rakenduse loomiseks kasutatakse IntelliJ IDEA sisemist võimekust, mis võimaldab otse arenduskeskkonnast kasutada Spring Initializr tööriista. Spring Initializr on veebirakendus, mis genereerib tühjad Spring Boot projektid [67].

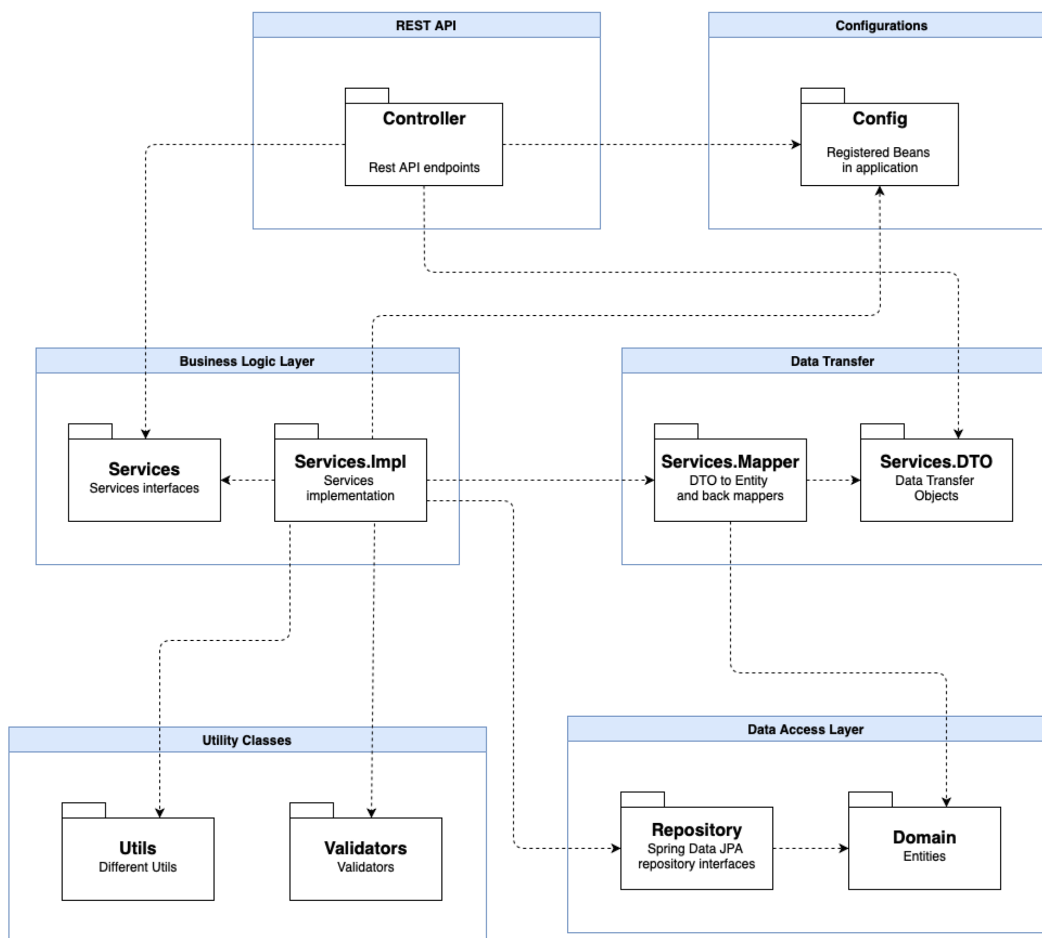


Joonis 5. Projekti loomine IntelliJ IDEA keskkonnas.

Joonisel 5 on näidatud, kuidas luua IntelliJ IDEAs uus Spring Boot, kasutades Spring Initializr'i võimekust. Joonisel näeme, et kasutatud on 2020 oktoobri seisuga viimast Java versiooni 15 [68]. Projekt luuakse Maven-tüüpi. Maven on projektijuhtimise tööriist, mis tegeleb projekti ülesehituste, sõltuvuste, levitamise, väljalasete jms lahendamisega [69]. Spring Booti sisse on juba integreeritud Tomcat [70] ja Jetty [71] serverid, seega lokaalselt masinas Spring Booti aplikatsiooni käivitamiseks ei ole eraldi vaja lisada või konfigureerida servereid [72].

Peale kirjeldatud sammude võtmist, luuakse uus projekt, mis sisaldab erinevaid komponente arendusega alustamiseks. Rakendus läheb kohe käima, sest on olemas peamine meetod (*main method*).

4.1.2 Programmi struktuur



Joonis 6. Serverrakenduse kihtide mudel (autori looming).

Joonisel 6 on välja toodud programmisisesed kihid. Iga kiht on eraldiseisev komponent, mis töötab kogu programmi informatsiooniga ja teiste kihtidega vastavalt sellele, milline on andmete manipuleerimise järjekord. Tavaliselt töötab iga kiht selle kohal ja selle all oleva kihiga [73]. Järgmiselt vaadeldakse igat kihti eraldi.

- Kontrollerite (*REST API*) kiht – sellest kihis on APIga seotud kontrollerid.
- Konfiguratsiooni (*Configuration*) kiht – asuvad kõik klassid ja failid, mis on vajalikud rakenduse konfigureerimiseks.
- Äriloogika (*Business Logic*) kiht – teenused, mis täidavad ärinõudmisi.
- Andmeedastuse (*Data Transfer*) kiht – klassid ja objektid, mis on seotud andmete edastusega. Siia alla kuuluvad ka tööriistad, mis konverteerivad andmemudeli andmeedastusobjektideks.

- Andmete juurdepääsu (*Data Access*) kiht – andmete juurdepääsu adapterid koos repositooriumitega ja andmemudelitega (*Entity*).
- Utiliitide (*Utilities*) kiht – asetsevad kõik abistavad utiliidid nagu näiteks validaatorid.

Kõik klassid on jaotatud omaette pakettidesse vastavalt ühe või teise klassi eesmärgile.

- *Config (*konfiguratsioon*)* – siin asuvad kõik klassid, mis on mõeldud Spring Booti aplikatsiooni konfigureerimiseks. Sellised klassid on annoteeritud *@Configuration* [74] annotatsiooniga. Siin all asuvad Retrofit, turvalisuse, parsimisvahendite ja muud konfiguratsioonid.
- *Constants (*konstandid*)* – kõik Java staatilised ja globaalsed muutujad ning Enum tüüpi klassid.
- *Service (*teenus*)* – pakett, kus asuvad teenused, DTO ja kaardistajad *(mapper)* ning samuti teenuste liidesed *(interface)*. Teenuste klassid on annoteeritud *@Service* annotatsiooniga.
- *Controller (*kontroller*)* – kuuluvad REST API kontrollerid. Antud klassid on annoteeritud *@RestController* annotatsiooniga.
- *Repository (*repositoorium*)* – kuuluvad kõik klassid ja liidesed vajalikud andmebaasiga suhtlemiseks. Repositooriumi all mõeldakse salvestamise, otsimise ja otsingukriteeriumite kapseldamise mehhanismi [75].
- *Domain (*domeen*)* – kõik *@Entity* tüüpi objektid. See on mistahes eraldiseisev identifitseeritav objekt [76].

Kogu projekti vältel on jälgitud ülaltoodud pakettide loogikat ning Java klassid on paigutatud vastava paketti alla. See võimaldab kiiremini üles otsida programmikoodi osad kõikidele programmi arendajatele.

4.1.3 Kasutatavad sõltuvused

Paberivaba lahenduse serverirakenduse projekt kasutab Maveni tööriista. Üks Maveni võimalustest on erinevate sõltuvuste juhtimine. Kõik vajaminevad paketid koos versioonidega on kirjeldatud POM failis, mis asub projekti juurkaustas. Seega selleks, et lisada uus sõltuvus *(dependency)* projekti juurde, tuleb ainult see kirjeldada POM failis ning IntelliJ IDEA koostöös Maveniga laadib selle alla ning lisab projekti juurde. Peale seda saab soovitud sõltuvust kohe koodis kasutada.

```xml
<properties>
    <java.version>15</java.version>
    <lombok.version>1.18.14</lombok.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
            <exclusion>
                <groupId>org.junit.vintage</groupId>
                <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

Joonis 7. Maven sõltuvuste lisamine.

Joonisel 7 on näidatud, kuidas saab POM failis lisada uusi sõltuvusi. Kõik sõltuvused lähevad *„dependency"* märksõna vahele. Kohustuslik on kirjeldada *groupId* ja *artifactId* väljad. Versiooni väli võib olla tühi, kui see sõltub teisest sõltuvusest. Praegusel juhul kasutatakse sama versiooni, mida kasutab Spring Boot ise. Leidub ka „*scope*" märgis, mis aitab piirata sõltuvuste transitiivsust [77]. Transitiivsuse teooria ütleb, et kui *a* on seotud *b* -ga ja *b* on seotud *c* -ga, siis on ka *a* seotud *c* -ga.

Järgnevalt on välja toodud kõige tähtsamad serverirakenduses kasutatavad sõltuvused:

- *Spring-boot-starter-data-jpa* – Spring Data JPA on üks osa Spring Data perekonnast. Moodul hõlbustab selliste Spring aplikatsioonide loomist, mis kasutavad andmete juurdepääsu tehnoloogiaid [78].

- *Spring-boot-starter-websocket* – teek, mis lisab kõik vajalikud komponendid, et kasutada *WebSocket* tehnoloogiat. WebSocket on kahesuunaline ja püsiv ühendus veebibrauseri ja serveri vahel [79].
- *Retrofit* – tüübikindel HTTP klient [80].

Kasutuses on ka palju teisi sõltuvusi. Lisas 2 on välja toodud kogu POM faili struktuur ja kõik kasutatavad sõltuvused.

4.1.4 Serverrakenduse konfiguratsioon

Spring Booti aplikatsiooni konfigureerimiseks on kasutatud Java põhilist konfigureerimist. See tähendab, et kogu konfiguratsioon tehakse Java klassis. Seejuures on klass annoteeritud erilise *@Configuration* annotatsiooniga [81]. Samuti on klassides võimalik defineerida erinevaid *@Bean* tüüpi objekte. Spring Booti mõistes on *bean* objektid, mis moodustavad aplikatsiooni selgroo ning mida haldab Spring ise [82].

```
@Bean("signingServiceRetrofit")
public Retrofit retrofit() {
    HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
    interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
    OkHttpClient client = new OkHttpClient.Builder()
            .addInterceptor(interceptor)
            .build();

    return new Retrofit.Builder()
            .baseUrl(signingServiceUrl)
            .addConverterFactory(JacksonConverterFactory.create(this.objectMapper))
            .client(client)
            .build();
}
```

Joonis 8. Retrofit konfiguratsioon.

Joonisel 8 on näidatud, kuidas toimub Retrofit HTTP kliendi konfigureerimine. Konfiguratsioon on vajalik, kuna võimaldab serverrakenduse programmil teha päringud väliste teenuste vastu. Antud näites luuakse uus Retrofit konfiguratsioon allkirjastamise teenusega suhtlemiseks. Kasutuses on *@Bean* annotatsioon, et märkida, et loodav objekt on kättesaadaval kõikidele teistele klassidele läbi sõltuvuse süstimise *(dependency injection)*. Märkida tuleb ka seda, et Retrofit kliendile lisatakse kuulajad *(interceptor)* selleks, et oleks võimalik logidesse kirjutada, millised päringud milliste andmetega tehakse ja milline informatsioon tuleb tagasi. Lisa 3 sisaldab kogu konfiguratsiooniklassi näidet.

```java
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
                .setAllowedOrigins("*");
    }
}
```

Joonis 9. WebSocket konfiguratsioon.

Joonisel 9 on näidatud WebSocketi konfiguratsiooni Spring Booti aplikatsioonis. Märkimisväärne on siin @EnableWebSocketMessageBroker annotatsioon, mis aktiveerib sõnumivahetuse funktsionaalsuse. Antud konfiguratsiooniklassi täielik versioon on saadaval lisa 4.

Serverrakenduse programmis leiduvad ka teised konfiguratsiooniklassid. Autori arvamusel ei ole nende kirjeldamine selles töös vajalik, sest nad katavad põhilist Spring Booti funktsionaalsust. Suurem osa konfiguratsioonidest oli tehtud *application.properties* failis. Kui Spring programm käivitub, loetakse kõik seaded antud failist sisse ning lisatakse need Spring *Environment* alla, kust hiljem on võimalik neid sätteid kätte saada.

4.1.5 REST API

Kõik REST APIga seotud klassid on annoteeritud *@RestController* annotatsiooniga. Järgnevalt näitan, millised API punktid loodi ja millisel eesmärgil. Antud juhul ei kirjeldata vastuse koode. Eeldatakse, et eduka protsessi vastuse kood on 200. Juhul, kui programmi töös tekib viga, siis vastavalt veale tagastatakse kood 400 või 500. Koodi 400 kasutatakse juhul, kui tekkis probleem andmete valideerimisega. Näiteks kui andmed on valed või puudulikud. Koodi 500 kasutatakse juhul, kui tekkis programmisisene probleem, mille puhul töö jätkamine ei ole võimalik. Sellisteks võib olla näiteks vale konfiguratsioon, vale andmete käitlus ja palju muud.

URI	Protokoll	Eesmärk
/api/handshake	GET	Tagastab kõik andmebaasis salvestatud kaardistused kliendinõustaja ja kliendi arvuti vahel.
/api/handshake/{host}	GET	Kasutades parameetrit *host,* leiab andmebaasist arvutite kaardistuste alt antud arvutinime paarilise.
/api/handshake	POST	Lisab uue kaardistuse andmebaasi.
/api/handshake/{id}	PUT	Uuendab andmebaasis kaardistuse saadetud muutuja *id* järgi.
/api/handshake/{id}	DELETE	Kustutab kaardistuse andmebaasist *id* järgi.
/api/commercial/active	GET	Tagastab andmebaasist praegu aktiivse reklaami olemi koos failiga.
/api/commercial/{id}	GET	Tagastab andmebaasist reklaami olemi parameetri *id* järgi.
/api/commercial/payload/{id}	GET	Tagastab andmebaasist reklaami videofaili base64 formaadis reklaami *id* järgi.
/api/commercial	POST	Salvestab uue reklaami olemi andmebaasi.
/api/commercial/{id}	PUT	Uuendab andmebaasis reklaami olemit *id* järgi.
/api/commercial/{id}	DELETE	Kustutab andmebaasis reklaami olemi *id* järgi.
/api/event/{type}	POST	Vastavalt saadetud *type* muutujale saadetakse läbi WebSocketi kliendi arvutisse käsk. Päringuga tulevad kaasa ka andmed päringu kehas. Need andmed töödeldakse ja saadetakse samuti kliendi arvutisse.
/actuator/health	GET	Tagastab 200 koodi, kui aplikatsioon töötab.

Tabel 1. REST API kirjeldus.

4.1.6 Ühendus andmebaasiga

Ühendus andmebaasiga on teostatud kasutades Spring Data JPA teegi võimalusi. Tänu sellele sõltuvusele toimub andmebaasi ühenduse konfigureerimine väga lihtsalt. Selleks kasutatakse varem mainitud *application.properties* faili.

```
# Database configurations
spring.datasource.url=jdbc:postgresql://localhost:5432/paperfree
spring.datasource.username=paperfree
spring.datasource.password=***
spring.datasource.initialization-mode=never
```

Joonis 10. Andmebaasi ühenduse loomine.

Joonisel 10 on näidatud, kuidas konfigureerida andmebaasi ühendus läbi *application.properties* faili lokaalse andmebaasi vastu. Iga rida failis on uus seadistus. URLis täpsustatakse, et ühendus luuakse vastu PostgreSQL andmebaasi kasutades vastavat draiverit. Viimane rida näites ütleb, et ei ole vaja genereerida andmebaasi tabeleid programmi kirjeldatud olemite järgi. Seega kõik tabelid ja nende kirjeldused luuakse andmebaasis käsitsi. Autori arvates ei taga sellise võimaluse kasutamine piisavat kontrolli andmebaasi struktuuri osas.

4.1.7 WebSocketi kasutamine

WebSocket on selles projektis üks keskseid tehnoloogiaid, sest võimaldab saata kliendi arvutile sõnumeid just sellel momendil, kui teine rakendus seda ütleb. Kui paberivaba serverirakendus saab uue käsu, saadetakse see käsk edasi otse kliendirakendusse. Järgnevalt vaatlengi, kuidas on tehtud sõnumite saatmine kliendi rakendusse WebSocketi tehnoloogiaga.

Spring Booti aplikatsioonis on WebSocketi kasutamine väga lihtne tänu sõltuvusele, mis eelnevalt programmi juurde lisati. Ainus mida on vaja teha, on lisada teenuse juurde klass nimega *SimpMessagingTemplate*, mille instants süstitakse (*dependency injection*) teenusesse.

```
private final SimpMessagingTemplate simpMessagingTemplate;
private final HandshakeService handshakeService;

public EventServiceImpl(SimpMessagingTemplate simpMessagingTemplate,
                        HandshakeService handshakeService) {
    this.simpMessagingTemplate = simpMessagingTemplate;
    this.handshakeService = handshakeService;
}
```

Joonis 11. Objekti süstimine teenusesse.

Joonisel 11 on näidatud, kuidas on realiseeritud *SimpleMessagingTemplate* objekti süstimine *EventServiceImpl* klassi kasutades seejuures süstimist läbi klassi konstruktori.

```
@SneakyThrows
private void dispatchEvent(Event event) {
    simpMessagingTemplate.convertAndSend(event.getDestination(), event);
}
```

Joonis 12. Sõnumi saatmine läbi WebSocket.

Joonis 12 näitab, kuidas toimub sõnumi saatmine kliendirakendusse. Vajalik on ainult määrata, milline on URL kuhu sõnum saadetakse. Antud URLis sisaldub ka kliendi arvutinimi, mis võimaldab otse sellele arvutile sõnumi saata.

See meetod on kirjeldatud eraldi teenuses. Teenus oli loodud selleks, et selle kaudu saata sõnumid kliendirakendusse. Enne saatmist tehakse viimased kontrollid, et kõik vajalikud andmed oleks kogutud. Vastasel juhul visatakse välja erand *(exception)*. Kogu teenuse implementatsioon koos selles kasutatavate klassidega on saadaval lisas 4.

4.2 Kliendi rakenduse arendus

Kliendirakenduse jaoks otsustati kasutada Angulari raamistiku, mis toetab Typescripti programmeerimiskeelt. Järgnevalt vaatlen, kuidas loodi uus Angulari projekt, milline on selle struktuur, kuidas on konfigureeritud WebSocketi kuulajad ja näiteks on toodud kasutajaliidese disain.

Rakendus vastutab ainult informatsiooni näitamise eest. Kliendirakendus ei talleta informatsiooni. See on tihedas koostöös Paberivaba serverrakendusega. Nende suhtlus käib läbi REST API ning kõik andmed talletatakse ja töödeldakse ainult Spring Booti aplikatsioonis.

4.2.1 Angulari rakenduse loomine

Rakenduse loomiseks on kasutatud Angular CLI programmi. See on käsurea liides, mis võimaldab lihtsalt luua uut aplikatsiooni, mis juba toimib ning jälgib kõige paremaid tavasid [83].

```
npm install -g @angular/cli
```

Joonis 13. Angular CLI installeerimine.

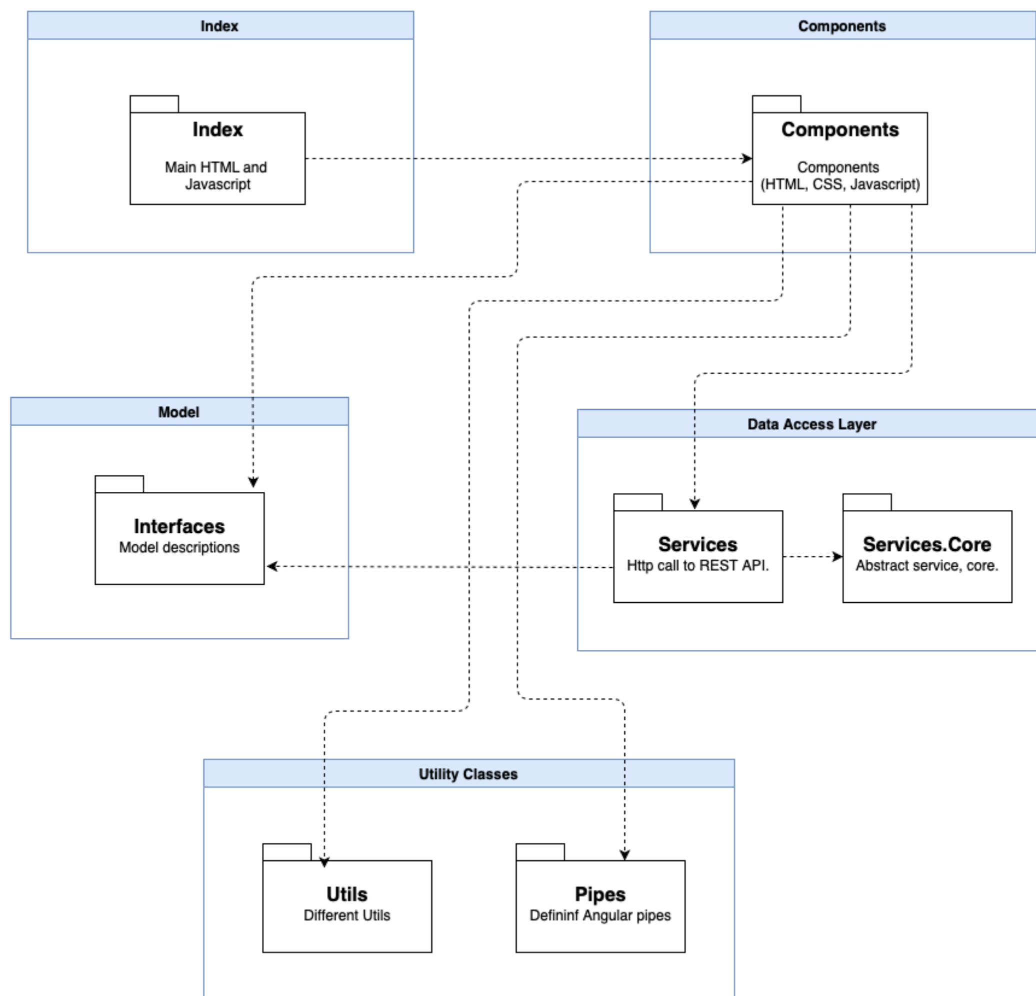
Joonisel 8 on näha Angular CLI käsk, mida tuleb käivitada käsurealt. Eelduseks on, et arvutis on installeeritud Node.js tarkvara ja sellega koos paigaldatav NPM *(Node Package Manager)*. See on repositoorium, kus avaldatakse avatud lähtekoodiga Node.js projektid. NPM on käsurea utiliit nimetatud repositooriumiga suhtlemiseks, mis võimaldab erinevate pakettide paigaldamist, versiooni ja sõltuvuste *(dependency)* haldust [84].

```
ng new paperfree-client
```

Joonis 14. Uue projekti loomine kasutades Angular CLI-d.

Kui Angular CLI tööriist on paigaldatud, on võimalik luua uus Angulari projekt. Joonisel 9 on näidatud terminali käsk, mis loob uue Angulari projekti nimega „*paperfree-client*“. Programm luuakse samasse direktooriumisse, kus antud käsk käivitati. Kui tööriist on oma töö lõpetanud, võib avada projekti läbi IntelliJ IDEA.

4.2.2 Programmi struktuur



Joonis 15. Kliendirakenduse komponentide jaotus ja nende seosed (autori looming).

Joonisel 10 on näidatud, kuidas jaotuvad erinevad Angulari projekti komponendid ning mis osa millega on seotud.

Komponent *(Components)* paketi all asuvad kõik loodavad Angulari komponendid. Angulari raamistiku mõistes on komponent koodi loogiline osa, mis koosneb mitmest erinevast osast. Esiteks mallist *(template)*, mille ülesanne on rakenduse vaate renderdamine. See osa sisaldab HTMLi koodi ning erinevaid siduvaid elemente. Teiseks osaks on klassid *(class)*, mis sisaldavad meetodeid ja muutujaid *(properties)* [85]. Lisaks eelnimetatud osadele on komponendiga seotud ka test ning on võimalik eraldi luua CSS

dokument, mis seostatakse hiljem komponendi klassis. Komponendi saab tuvastada @Component annotatsiooni järgi [86].

Joonisel 10 on eraldatud andmete juurdepääsu kiht *(Data Access Layer)*, mille sees asuvad teenused ning baasteenus, mida kasutavad kõik teised teenused. Baasteenuses on kirjeldatud kõik meetodid, mis on vajalikud, et teha päring vastu *Paperfree* serverrakendust. Teenustes on kirjeldatud ainult mis andmeid oodatakse, millist URLi on vaja kasutada, et nendele ligi pääseda ning milliseid andmeid on vaja koos päringuga kaasa anda. Sellised klassid on annoteeritud *@Service* annotatsiooniga. Teenuse erinevus on selles, et seda saab süstida teistesse, näiteks komponentide klassidesse [87].

Abistavad klassid on samuti jaotatud eraldi pakettidesse. Mudeli *(model)* paketi all asuvad liidesed *(interface)*, mis kirjeldavad andmete struktuuri. See on vajalik selleks, et kontrollida, kas sisse tulevad andmed jälgivad teatud struktuuri. Erinevad utiliidid on samuti kogutud ühe paketi alla, et projekt oleks paremini struktureeritud ja koodijuppide leidmine oleks kiirem ja lihtsam.

```
ng generate component components/commercial
ng generate service services/commercial

ng serve
```

Joonis 16. Komponentide, teenuste loomine ja programmi käivitamine kasutades Angular CLI-d.

Joonisel 11 on välja toodud teised Angular CLI võimalused. Esimesed kaks rida näitavad, kuidas on võimalik kiiresti ja mugavalt luua uus komponent ja uus teenus. Peale käsu käivitamist genereeritakse vastavasse direktooriumisse kõik vajalikud failid ja lisatakse annotatsioonid. Viimane käsk näitab, kuidas saab lokaalselt käivitada Angulari programmi.

Lahenduse kõik sõltuvused on kirjeldatud *package.json* failis. NPM kasutab hiljem seda faili, et installeerida vajalikud välised paketid, mis on kirjeldatud selle all. Kui programm luuakse, genereeritakse ka *package.json* dokument, mis sisaldab kohe nii öelda

stardipaketti erinevatest sõltuvustest. Mõned on vajalikud Angulari toimimiseks, teised aga katavad levinumaid stsenaariume [88].

4.2.3 Päringute tegemine vastu serverrakendust

Kuna kliendirakendus töötab koostöös serverrakendusega, siis peab olema võimalik teha päringuid vastu serverrakendust, et sealt informatsiooni kätte saada. Oli võetud kasutusele selline süsteem, mille puhul kogu andmete päring toimib läbi teenuse, mis on jaotatud vastavalt päritavatele andmetele. Nii näiteks asuvad ühe teenuse all kõik päringud, mis on vajalikud reklaamide kättesaamiseks.

Kõik teenused kasutavad baasteenust, kus on kirjeldatud kõiki päringu protokolle ja meetodeid, kuidas tuleb töödelda vastusena saadetud andmeid. Antud baasteenus on leitav lisas 5. Teised teenused kasutavad baasteenusest kirjeldatud meetodeid, et uus päring vastu REST APIt teha.

```
public getActiveCommercial(): Subscription {
    this.commercialPayload = null;
    return this._http.get('/commercial/active/')
        .pipe(
            map((data: any) => {
                this._log.info('Received active commercial payload');
                if (data === null) {
                    return of(null);
                }
                this.commercialPayload = data;
                return of(data);
            })).subscribe((data: any) => {
            if (this.commercialPayload !== null || data) {
                this._decoder.decodeCommercial(this.commercialPayload)
                    .subscribe((value) => {
                        if (value !== false) {
                            this.commercialVideo = value;
                        }
                    });
            }
        });
}
```

Joonis 17. HTTP päringu tegemine Angularis.

Joonisel 17 on näidatud, kuidas tehakse HTTP GET päring serverrakenduse vastu, et saada kätte praegu aktiivne reklaam koos failiga. Näites on kasutatud *_http* muutuja, mille sees on baas HTTP teenus.

Kõik muud päringud vastu serverrakendust on tehtud sama analoogia järgi. Teenustes kasutatakse palju logimist, et oleks võimalik lihtsamini aru saada, kust probleem tuleb. Logi võimaldab teenustes näha, kas päring oli edukas ning millised andmed saabusid.

4.2.4 WebSocketi kuulaja kliendirakenduses

Selleks, et Angular programm saaks kätte serverirakendusest talle saadetud sõnumid, tuleb alguses konfigureerida selle kuulaja. Kuulaja konfigureerimisel peab olema teada, millisel arvutil antud kliendirakendus jookseb ehk tuleb teada arvuti nime. Eeldatakse, et see teadmine on juba eelnevalt olemas. Samuti eeldatakse, et on konfigureeritud ja teada serverrakenduse URL.

```
const stompConfig: InjectableRxStompConfig = Object.assign({}, data, {
    brokerURL: data.brokerURL,
    beforeConnect: () => {
        console.log('%c called before connect', 'color: blue');
    },
    debug: (msg: string): void => {
        console.log(new Date(), msg);
    }
});

this._rxStompService.configure(stompConfig);
this._rxStompService.activate();
this._watchIncomingMessages();
```

Joonis 18. WebSocketi konfiguratsioon kliendirakenduses.

Joonis 18 näitab, kuidas konfigureeritakse teenus, mis hakkab kuulama sissetulevaid sõnumeid. Selleks on kasutatud *RxStomp* teenus, mis tuleb *„ng2-stompjs“* teegist, mis oli eelnevalt lisatud *package.json* faili. Antud teek võimaldab teha STOMPi WebSocketi kliendi kaudu [89].

```
private _watchIncomingMessages(): void {
        // We suppose, that current method is returning correct computer name.
        const hostname = this._getHostName();
        this._rxStompService.connected$.subscribe((data: any) => {
            this._rxStompService.watch('/queue/' + hostname).subscribe((message:
Message) => {
                const receivedEvent = <PaperfreeEvent>JSON.parse(message.body);

                this._log.info('New event received', receivedEvent);

                if (!Object.values(EventType).includes(receivedEvent.type)) {
                    this._log.warn('Received unsupported event',
receivedEvent.type);
                } else {
                    // Service, which is processing incoming data.
                    this._incomingMessageService
                    .processReceivedEvent(receivedEvent.type, message.body);
                }
            });
        });
}
```

Joonis 19. Sõnumite kuulamise näide.

Joonisel 19 on näidatud, kuidas on teostatud sõnumite kuulamine. Selleks hakatakse kuulama URI, mis sisaldab arvuti nime. Kui uus sõnum on tulnud, siis muudetakse JSON objektiks, kasutades selleks eelnevalt defineeritud liidest. Järgmisena kontrollitakse, kas saadud käsu tüüp on toetatud. Kui ei ole, siis lisatakse vastav märge logidesse. Vastasel juhul saadetakse kogu sõnum edasi teenusele, mis hakkab seda protsessima ja vajalikke programmi osasid käivitama.

See konfiguratsioon ja kuulaja on registreeritud kliendirakenduse komponendis, millest algab kogu programmi elutsükkel. Seepärast on garanteeritud, et komponent käivitatakse ja kuulaja saavad registreeritud.

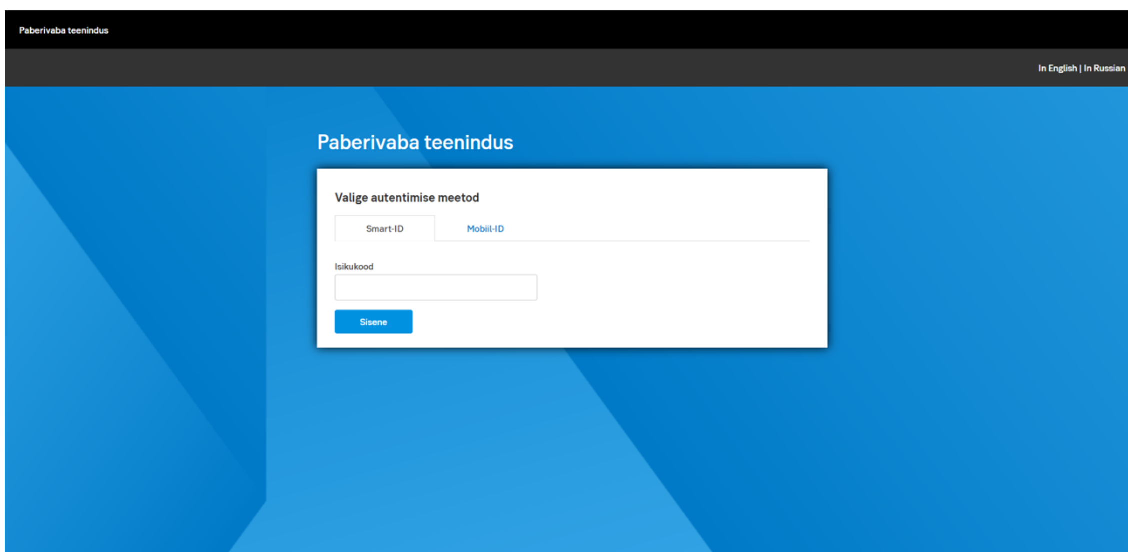
4.2.5 Kasutajaliidese välimus

Rakenduse disainimisel oli eesmärgiks luua lihtne ja arusaadav kasutajaliides, mille välimus on kaasaegne ja ilus. Rakenduse välimuse disainis konsulteeriti kasutajakogemuse ehk UX/UI spetsialistiga, kes visandas, kuidas rakendus võiks välja näha. Kuna ei olnud ühtegi taolist rakendust eeskujuks, kujunes välimus autori nägemuse ja kasuajakogemuse spetsialistide soovituse järgi.

Värvideks valiti sinine, valge ja hall, mis mõjuvad inimesele lõõgastavalt ja rahustavalt. Sinine on rahu ja harmoonia [90]. Hall värv on kasutuses rohkem kui aktsentvärv. Värvivalik oli paljuski mõjutatud just autori enda soovidest.

Disaini arendus põhineb Bootstrapi teegil ja teegi pakutavatel komponentidel. Bootstrap on tasuta raamistik, mis võimaldab kiiresti luua dünaamilist veebidisaini. See on praegu üks populaarsemaid disainiteeke maailmas. Ta sisaldab dünaamilisi võrgusüsteeme *(grid)*, komponente ja erinevaid Javascripti liideseid [91]. Tuleb märkida, et Bootstrap võimaldab ka väga head dokumentatsiooni, mis võimaldab näha, kuidas komponent hakkab välja nägema.

Eeldus oli, et kasutajaliides peab hea välja nägema igal ekraanil. Arvestati, et rakendust võib avada mobiilis, arvutis või tahvelarvutis. Kasutades Bootstrapi dünaamilist võrgusüsteemi ei olnud ülesanne raske. Ainuke tingimus oli täpselt jälgida dokumentatsiooni.



Joonis 20. Kliendi autentimise vaade.

4.3 Testimine

Lõputöös valminud lahenduse testimiseks kasutatakse manuaalset testimist. Manuaalset testimist rakendatakse iga kord, kui on arendatud uus funktsionaalsus. Programmi testimine toimub kahes osas – serveri rakenduse ja kliendi rakenduse testimine.

Serveri rakenduse testimiseks kasutatakse Postmani aplikatsiooni, mis on tasuta saadaval nende veebilehel. Seda rakendust võib kasutada API testimiseks. Kuna serverrakendusel puudub kasutajaliides, siis Postman võimaldab saata manuaalselt päringuid vastu REST APIt ning vaadata, millega Paberivaba *backend* rakendus vastab. Tuleb märkida, et kasutades Postmani funktsionaalsust on võimalik luua ka automaatsed testid, mis valideeriksid tagasitulevaid andmeid. Selles töös ei vaadelda selliste skriptide loomist.

Kasutajaliidese ja kliendirakenduse äriloogika testimine toimub samuti manuaalselt. Testija avab kliendirakenduse, kus proovitakse erinevat funktsionaalsust ja vaadeldakse, kas rakendus vastab nõuetele. Samuti mängitakse läbi erinevaid stsenaariume ning kontrollitakse, et protsessis ei tekiks vigu.

Kasutatakse ka kombineeritud manuaalset testimist, mille puhul saadetakse Postmani kaudu REST API vastu päring käsuga kuvada informatsioon kliendi ekraanile. Selle testi puhul peaks kliendirakenduse ekraanile ilmuma andmed, mis olid eelnevalt saadetud Postmani aplikatsioonist. See võimaldab kontrollida, kas ekraani jagamine toimib.

Rakendust vaatavad üle esiteks arendajad. Nemad kasutavad Postmani rakendust, et kontrollida, kas nende loodud API vastab nõuetele. Programmeerijad teevad kindlaks, kas kliendirakendus toimib õigesti ja ei teki vigu. Järgmiseks antakse rakendus proovida väiksemale äriinimeste kogukonnale, kes kirjeldasid nõudeid ning kes oskavad üle vaadata, kas rakendus vastab kõikidele ärinõuetele. Tähtsamad testid, mida tehakse, on järgmised:

- Kas kliendirakendus avaneb ja reklaamipildid kuvatakse
- Kas klient saab ennast autentida kasutades digitaalseid meetodeid. Vaadatakse üle, kuidas programm käitub juhul, kui klienti ei leitud, kui oli sisestatud vale PIN ning kui kasutaja ei sisestanud midagi.
- Kas toimib ekraanijagamine. Siinkohal valideeritakse, et ekraani jagamist ei tohi toimuda juhul, kui klient ei ole ennast varem autentinud.
- Kas edasisuunamine allkirjastamise lehele toimib. Juhul, kui kõik dokumendid ja kõik osapooled on oma allkirja andnud, kas toimub tagasisuunamine

kliendirakendusse ning kas kuvatakse õige teade. Juhul, kui allkirjastamise käigus tekkis probleem peab kliendirakendus sellest kliendile teada andma.

- Vaadatakse, et oleks loodud digiümbrik ning kõik allkirjad ja dokumendid korrektselt kogutud.

5 Kokkuvõte

Bakalaureusetöö käigus valmis jätkusuutlik klienditeeninduse protsess, kus klient on kaasatud teenindusse ja lepingute allkirjastamine ning arhiveerimine on digitaalne. Töötajal on võimalik kuvada lepingu infot kliendile arvuti ekraanile. See võimaldab kliendil jälgida kogu teenindusprotsessi jooksul töötaja poolt sisestatud informatsiooni ja vajadusel paluda teenindajal seda parandada. Pärast lepingu andmete kontrolli allkirjastatakse dokument nii kliendi kui ka pangatöötaja poolt digitaalselt. Pärast pangatempli andmist salvestatakse digitaalne konteiner panga andmebaasi. Kliendile on leping kättesaadav internetipangas või vajadusel saab saata dokumendi e-posti aadressile.

Kuna digitaliseerimine on innovatsiooni ja ettevõtlusega tihedalt seotud, siis loodud lahendus aitab kaasa ettevõtte jätkusuutlikule arengule. See omakorda vähendab paberi käitlemisega seotud kulusid ja minimiseerib lepingute haldusega tekkivaid riske. Samuti annab see võimaluse töötajatele tegeleda ettevõttele rohkem kasutoovate tegevustega, kuna nemad ei pea enam tegelema dokumentide arhiveerimise ja kontrolliga.

Lisaks ettevõtte positiivsele majanduslikule mõjule on uudne tööprotsess loodust säästev ja tõstab ettevõtte sotsiaalset vastutust. Kasutades vähem paberit, raiutakse vähem puid, mis omakorda säästab meie looduskeskkonda.

Uus lahendus pakub erinevaid jätkusuutlikke kasutusvõimalusi. Näiteks võib süsteemi rakendada videonõustamise keskkonnas, kus klient saab oma asukohas pangaga suhelda ja lepinguid allkirjastada. See vähendab kliendi aja- ja transpordikulusid. Pank muutub kliendi jaoks paindlikumaks ja kättesaadavamaks. Kuna dokumendid on digitaalsed, on võimalik luua uudne süsteem lepingute väljaotsimiseks ja sorteerimiseks. Seega ei pea enam ootama lepingu koopiat arhiivist mitu päeva, vaid saab selle kätte minutitega.

Autor arvab, et digitaliseerimise ajastul tuleb kasutada kõiki võimalusi, et olla loodussõbralik, jätkusuutlik ning pakkuda kliendile kiiret, efektiivset ja interaktiivset lahendust traditsiooniliste töövõtete asemele.

Kasutatud kirjandus

[1] Trinidad Wiseman, [Võrgumaterjal]. Available: https://atlassian.twn.ee/atlassiani-tooted/. [Kasutatud 19 Oktoober 2020].

[2] „DevOps Tutorial: Complete Beginners Training," Guru99, [Võrgumaterjal]. Available: https://www.guru99.com/devops-tutorial.html. [Kasutatud 19 Oktoober 2020].

[3] „Java Enums," W3Schools, [Võrgumaterjal]. Available: https://www.w3schools.com/java/java_enums.asp. [Kasutatud 19 Oktoober 2020].

[4] „Dokumendipank," [Võrgumaterjal]. Available: http://www.dokumendipank.ee/sailitustahtajad/. [Kasutatud 3 Oktoober 2020].

[5] „Internal data," SEB Eesti AS. [Võrgumaterjal].

[6] „Conservatree - Trees Into Paper," [Võrgumaterjal]. Available: http://www.conservatree.org/learn/EnviroIssues/TreeStats.shtml. [Kasutatud 3 Oktoober 2020].

[7] „Riigi Teataja," Riigikogu, 8 Märts 2000. [Võrgumaterjal]. Available: https://www.riigiteataja.ee/akt/694375. [Kasutatud 4 Oktoober 2020].

[8] A. S. R. I. A. OÜ Smartlink, „DigiDoc3 Kliendi ja DigiDoc3 Krüpto kasutusjuhend," 1 Oktoober 2010. [Võrgumaterjal]. Available: https://www.id.ee/public/Digidoc_Klient_Krupto_est.pdf. [Kasutatud 4 Oktoober 2020].

[9] „SEB,“ [Võrgumaterjal]. Available: https://www.seb.ee/igapaevapangandus/teeninduskanalid/paberivaba-teenindus. [Kasutatud 8 Oktoober 2020].

[10] „Dokumendihaldustarkvara DELTA tutvustus,“ SMIT, [Võrgumaterjal]. Available: https://www.smit.ee/et/delta-dokumendihaldussuesteem. [Kasutatud 19 Oktoober 2020].

[11] „Delta,“ Registrite ja Infosüsteemide Keskus, [Võrgumaterjal]. Available: https://www.rik.ee/et/asutusest/delta. [Kasutatud 19 Oktoober 2020].

[12] „Delta,“ Bitbucket, [Võrgumaterjal]. Available: https://bitbucket.org/smitdevel/delta/wiki/Home. [Kasutatud 19 Oktoober 2020].

[13] S. Dhruv, „Pros and Cons of using PostgreSQL for Application Development,“ Aalpha, 15 Mai 2019. [Võrgumaterjal]. Available: https://www.aalpha.net/blog/pros-and-cons-of-using-postgresql-for-application-development/. [Kasutatud 10 Oktoober 2020].

[14] „Digisign-client,“ Bitbucket, [Võrgumaterjal]. Available: https://bitbucket.org/smitdevel/digisign-client/src/master/. [Kasutatud 19 Oktoober 2020].

[15] „Likumi - Legal Acts of the republic of Latvia,“ 11 November 2002. [Võrgumaterjal]. Available: https://likumi.lv/ta/en/en/id/68521-electronic-documents-law. [Kasutatud 4 Oktoober 2020].

[16] „Latvija.lv,“ 7 Juuli 2019. [Võrgumaterjal]. Available: https://www.latvija.lv/en/DzivesSituacijas/tiesibu-aizsardziba/elektroniskais-paraksts#show2. [Kasutatud 4 Oktoober 2020].

[17] H. Lõugas, „DigiGeenius,“ 3 Märts 2017. [Võrgumaterjal]. Available: https://digi.geenius.ee/rubriik/uudis/smart-id-abil-saab-nuud-sisse-logida-swedbanki-ja-seb-panka/. [Kasutatud 4 Oktoober 2020].

[18] „Lietuvos Respublikos Seimas,“ 26 Aprill 2018. [Võrgumaterjal]. Available: https://e-seimas.lrs.lt/portal/legalAct/en/TAD/c5174772ecd011e89d4ad92e8434e309. [Kasutatud 4 Oktoober 2020].

[19] „SK Id Solutions,“ [Võrgumaterjal]. Available: https://www.skidsolutions.eu/en/about/history. [Kasutatud 4 Oktoober 2020].

[20] „Ministry of Transport and Communications, Finland,“ [Võrgumaterjal]. Available: https://www.finlex.fi/en/laki/kaannokset/2009/en20090617.pdf. [Kasutatud 4 Oktoober 2020].

[21] V. Vorteil, „Tarkvara analüüs ja testimine,“ [Võrgumaterjal]. Available: https://web.htk.tlu.ee/digitaru/testimine/chapter/tarkvara-arendusnouded/. [Kasutatud 6 Oktoober 2020].

[22] S. Butcher, „efinancialcareers,“ 15 Jaanuar 2020. [Võrgumaterjal]. Available: https://news.efinancialcareers.com/uk-en/3001980/programming-languages-jobs-banks-finance. [Kasutatud 5 Oktoober 2020].

[23] „Jetbrains,“ [Võrgumaterjal]. Available: https://www.jetbrains.com/lp/devecosystem-2020/?gclid=CjwKCAjwiOv7BRBREiwAXHbv3L2UyH6p8L1n3QF050JAK8GnuTssbutOI8lQ9Edlcg8hm3Ve7qcbjhoC-jgQAvD_BwE&gclsrc=aw.ds. [Kasutatud 5 Oktoober 2020].

[24] D. Yang, „Fullstack academy,“ [Võrgumaterjal]. Available: https://www.fullstackacademy.com/blog/nine-best-programming-languages-to-learn. [Kasutatud 5 Oktoober 2020].

[25] C. Martinez, „London APP Developer,“ November 2018. [Võrgumaterjal]. Available: https://londonappdeveloper.com/2018/11/08/traditional-vs-single-page-websites-and-which-language-to-learn/. [Kasutatud 5 Oktoober 2020].

[26] „Haridussilm," [Võrgumaterjal]. Available: https://www.haridussilm.ee/?leht=korg_4. [Kasutatud 10 Oktoober 2020].

[27] „The Python Wiki," Python, 16 September 2018. [Võrgumaterjal]. Available: https://wiki.python.org/moin/FrontPage. [Kasutatud 10 Oktoober 2020].

[28] A. Spittel, „What is a Web Framework, and Why Should I use one?," We Learn Code LLC, [Võrgumaterjal]. Available: https://welearncode.com/what-are-frontend-frameworks/. [Kasutatud 10 Oktoober 2020].

[29] „Meet Django," Django, [Võrgumaterjal]. Available: https://www.djangoproject.com. [Kasutatud 10 Oktoober 2020].

[30] „C# - Introduction," Microsoft, [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction. [Kasutatud 10 Oktoober 2020].

[31] „Everything you need to know about C#," Pluralsight, 15 Mai 2019. [Võrgumaterjal]. Available: https://www.pluralsight.com/blog/software-development/everything-you-need-to-know-about-c-. [Kasutatud 10 Oktoober 2020].

[32] „C#," Jetbrains, [Võrgumaterjal]. Available: https://www.jetbrains.com/lp/devecosystem-2020/csharp/?gclid=Cj0KCQjwt4X8BRCPARIsABmcnOrFukxIpjHoHZVrEWS4K13Tt9EGMrovRH5YglnzZ1hwtPSUjiNZ6xMaApX5EALw_wcB&gclsrc=aw.ds. [Kasutatud 10 Oktoober 2020].

[33] „What is ASP.NET Core," Microsoft, [Võrgumaterjal]. Available: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core. [Kasutatud 10 Oktoober 2020].

[34] „What Is PHP?," PHP, [Võrgumaterjal]. Available: https://www.php.net/manual/en/intro-whatis.php. [Kasutatud 10 Oktoober 2020].

[35] P. Andrews, „10 Popular PHP frameworks in 2020," Medium, November 2019. [Võrgumaterjal]. Available: https://morioh.com/p/45fce21d9a82. [Kasutatud 10 Oktoober 2020].

[36] R. Laanemets, „Veebirakenduste raamistikud: Struts, Spring ja JSF," 2005. [Võrgumaterjal]. Available: https://courses.cs.ut.ee/2005/tvt/uploads/Main/RaivoLaanemets_raamistikud.pdf. [Kasutatud 10 Oktoober 2020].

[37] E. Hanser, „Java Raamistikud," 2007. [Võrgumaterjal]. Available: https://courses.cs.ut.ee/2007/tvt/uploads/Main/se_07_07.pdf. [Kasutatud 10 Oktoober 2020].

[38] A. Suschevich, „10 Popular Java Frameworks," Medium, 8 Juuni 2020. [Võrgumaterjal]. Available: https://medium.com/javarevisited/10-popular-java-frameworks-for-web-applications-691c28f6c182. [Kasutatud 10 Oktoober 2020].

[39] K. Oha, „Spring Boot raamistik," 2017. [Võrgumaterjal]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjDzrbR6qnsAhXc8OAKHV-KDpMQFjAEegQIBBAC&url=http%3A%2F%2Fwww.cs.tlu.ee%2Fteemad%2Fget_file.php%3Fid%3D671&usg=AOvVaw2hw6SHXdF9ephQT6XQY1l2. [Kasutatud 10 Oktoober 2020].

[40] „Top Front-End frameworks in 2020," Existek, 14 Jaanuar 2020. [Võrgumaterjal]. Available: https://existek.com/blog/top-front-end-frameworks-2020/. [Kasutatud 10 Oktoober 2020].

[41] S. K. Arora, „10 Best JavaScript Frameworks to Use in 2020," hackr.io, 21 September 2020. [Võrgumaterjal]. Available: https://hackr.io/blog/best-javascript-frameworks. [Kasutatud 10 Oktoober 2020].

[42] J. Rumjantsev, „JavaScript raamistikude analüüs ja võrdlus struktureeritud SPA rakendustes," 2015. [Võrgumaterjal]. Available:

https://digikogu.taltech.ee/et/Download/951b32ac-3ab9-4fe2-ac46-ae481179611f. [Kasutatud 10 Oktoober 2020].

[43] A. Villems, „Andmebaasi Definitsioonid,“ [Võrgumaterjal]. Available: https://courses.cs.ut.ee/MTAT.03.264/2013_spring/uploads/Main/L01_Andmebaasi_definitsioonid.pdf. [Kasutatud 10 Oktoober 2020].

[44] „Mis vahe on relatsioonilise ja mitteseotud andmebaasi vahel,“ Strephonsays, [Võrgumaterjal]. Available: https://et.strephonsays.com/what-is-the-difference-between-relational-and-nonrelational-database. [Kasutatud 10 Oktoober 2020].

[45] P. Raspel, „Relatsioonilised andmebaasid,“ Andmebaaside alused, [Võrgumaterjal]. Available: https://enos.itcollege.ee/~priit/1.%20Andmebaasid/1.%20Loengumaterjalid/. [Kasutatud 10 Oktoober 2020].

[46] S. Barol, „PostgreSQL,“ IT College, [Võrgumaterjal]. Available: https://wiki.itcollege.ee/index.php/PostgreSQL. [Kasutatud 10 Oktoober 2020].

[47] „Oracle Database 19c,“ 4 Veebruar 2019. [Võrgumaterjal]. Available: https://www.oracle.com/a/tech/docs/database19c-wp.pdf. [Kasutatud 10 Oktoober 2020].

[48] „Mis vahe on SQL ja MySQL vahel,“ Stephonsays, [Võrgumaterjal]. Available: https://et.strephonsays.com/what-is-the-difference-between-sql-and-mysql#menu-2. [Kasutatud 10 Oktoober 2020].

[49] A. Mudianto, „MySQL Database Pros and Cons, Why Is It Really Popular,“ Techgalery, 7 November 2019. [Võrgumaterjal]. Available: https://www.techgalery.com/2019/11/mysql-database-pros-and-cons-why-mysql.html. [Kasutatud 10 Oktoober 2020].

[50] „Versioonihalduskeskkonnad GitHubi, GitLabi ja Bitbucketi näitel,“ [Võrgumaterjal]. Available:

https://wiki.itcollege.ee/index.php/Versioonihalduskeskkonnad_GitHubi,_GitLabi_ja_Bitbucketi_näitel. [Kasutatud 10 Oktoober 2020].

[51] J. v. Gumster, „6 places to host your git repository," Opensource.com, 30 August 2018. [Võrgumaterjal]. Available: https://opensource.com/article/18/8/github-alternatives. [Kasutatud 11 Oktoober 2020].

[52] „About," GitHub, [Võrgumaterjal]. Available: https://github.com/about. [Kasutatud 11 Oktoober 2020].

[53] „What Is GitHub? A Beginner's Introduction to GitHub," Kinsta, 2 Märts 2020. [Võrgumaterjal]. Available: https://kinsta.com/knowledgebase/what-is-github/. [Kasutatud 11 Oktoober 2020].

[54] „About," GitLab Inc., [Võrgumaterjal]. Available: https://about.gitlab.com/what-is-gitlab/. [Kasutatud 11 Oktoober 2020].

[55] „Jira Software," Atlassian, [Võrgumaterjal]. Available: https://www.atlassian.com/software/jira. [Kasutatud 11 Oktoober 2020].

[56] „Why BuitBucket," Atlassian, [Võrgumaterjal]. Available: https://bitbucket.org/product. [Kasutatud 11 Oktoober 2020].

[57] „Search all customer stories," Atlassian, [Võrgumaterjal]. Available: https://www.atlassian.com/customers/search?&search=&industry=financial-services. [Kasutatud 11 Oktoober 2020].

[58] „13 BEST Java IDE (2020 Update)," Guru99, [Võrgumaterjal]. Available: https://www.guru99.com/best-java-ide.html. [Kasutatud 11 October 2020].

[59] „What is Eclipse?," Educative.io, [Võrgumaterjal]. Available: https://www.educative.io/edpresso/what-is-eclipse. [Kasutatud 11 Oktoober 2020].

[60] „IntelliJ IDEA," Technopedia, [Võrgumaterjal]. Available: https://www.techopedia.com/definition/7755/intellij-idea. [Kasutatud 11 Oktoober 2020].

[61] D. Stern, „What Are WebSockets? A Brief Introduction," 10 November 2015. [Võrgumaterjal]. Available: https://code.tutsplus.com/tutorials/what-are-websockets-a-brief-introduction--cms-25239. [Kasutatud 12 Oktoober 2020].

[62] M. Rouse, „in-memory database," WhatIs.com, [Võrgumaterjal]. Available: https://whatis.techtarget.com/definition/in-memory-database. [Kasutatud 12 Oktoober 2020].

[63] „What are the best scalable In-Memory Databases?," Slant, [Võrgumaterjal]. Available: https://www.slant.co/topics/3237/~best-scalable-in-memory-databases. [Kasutatud 12 Oktoober 2020].

[64] „A performance evaluation of in-memory databases," ScienceDirect, [Võrgumaterjal]. Available: https://www.sciencedirect.com/science/article/pii/S1319157816300453. [Kasutatud 12 Oktoober 2020].

[65] „Objektorienteeritud programmeerimise põhimõtted," EUCIP, [Võrgumaterjal]. Available: https://eopearhiiv.edu.ee/e-kursused/eucip/arendus/342_objektorienteeritud_programmeerimise_phimtted.html. [Kasutatud 14 Oktoober 2020].

[66] „SOLID," TTU, [Võrgumaterjal]. Available: https://ained.ttu.ee/javadoc/oop/oop-solid.html. [Kasutatud 14 Oktoober 2020].

[67] C. Walls, „Initializing a Spring Boot project with Spring Initializr," [Võrgumaterjal]. Available: https://freecontent.manning.com/wp-content/uploads/initializing-a-spring-boot-project-with-spring-initializr.pdf. [Kasutatud 14 Oktoober 2020].

[68] „Java SE Downloads,“ Oracle, [Võrgumaterjal]. Available: https://www.oracle.com/java/technologies/javase-downloads.html. [Kasutatud 14 Oktoober 2020].

[69] „Mis vahe on Maven ja Gradle vahel,“ Stephonsays, [Võrgumaterjal]. Available: https://et.strephonsays.com/what-is-the-difference-between-maven-and-gradle. [Kasutatud 14 Oktoober 2020].

[70] K. Veskimäe, „Tomcat,“ 2013. [Võrgumaterjal]. Available: https://wiki.itcollege.ee/index.php/Tomcat. [Kasutatud 14 Oktoober 2020].

[71] „Tomcat vs Jetty,“ Dailyrazor, [Võrgumaterjal]. Available: https://www.dailyrazor.com/blog/tomcat-vs-jetty/. [Kasutatud 14 Oktoober 2020].

[72] „Building an Application with Spring Boot,“ Spring.io, [Võrgumaterjal]. Available: https://spring.io/guides/gs/spring-boot/. [Kasutatud 14 Oktoober 2020].

[73] „Program Layer,“ Techopedia, November 2012. [Võrgumaterjal]. Available: https://www.techopedia.com/definition/24812/program-layer. [Kasutatud 14 Oktoober 2020].

[74] „Spring @Configuration Annotation,“ JournalDev, [Võrgumaterjal]. Available: https://www.journaldev.com/21033/spring-configuration-annotation. [Kasutatud 15 Oktoober 2020].

[75] „Spring Boot @Repository,“ ZetCode, 6 Juuli 2020. [Võrgumaterjal]. Available: http://zetcode.com/springboot/repository/. [Kasutatud 15 Oktoober 2020].

[76] „Entity,“ Technopedia, [Võrgumaterjal]. Available: https://www.techopedia.com/definition/14360/entity-computing. [Kasutatud 15 Oktoober 2020].

[77] „Maven Dependency Scopes,“ Baeldung, 20 Juuli 2020. [Võrgumaterjal]. Available: https://www.baeldung.com/maven-dependency-scopes. [Kasutatud 15 Oktoober 2020].

[78] „Spring Data JPA," Spring, [Võrgumaterjal]. Available: https://spring.io/projects/spring-data-jpa. [Kasutatud 15 Oktoober 2020].

[79] „Intro to WebSockets with Spring," Baeldung, 10 September 2020. [Võrgumaterjal]. Available: https://www.baeldung.com/websockets-spring. [Kasutatud 15 Oktoober 2020].

[80] „Retrofit," [Võrgumaterjal]. Available: https://square.github.io/retrofit/. [Kasutatud 15 Oktoober 2020].

[81] R. Johnson, „A Java configuration option for Spring," Spring, 28 November 2006. [Võrgumaterjal]. Available: https://spring.io/blog/2006/11/28/a-java-configuration-option-for-spring/. [Kasutatud 17 Oktoober 2020].

[82] „Core Technologies," Spring, [Võrgumaterjal]. Available: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-definition. [Kasutatud 17 Oktoober 2020].

[83] „Angular CLI," Angular, [Võrgumaterjal]. Available: https://cli.angular.io. [Kasutatud 15 Oktoober 2020].

[84] „What is npm?," Node.js, 26 August 2011. [Võrgumaterjal]. Available: https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/. [Kasutatud 15 Oktoober 2020].

[85] „Angular 2 - Components," Tutorialspoint, [Võrgumaterjal]. Available: https://www.tutorialspoint.com/angular2/angular2_components.htm. [Kasutatud 15 Oktoober 2020].

[86] „Introduction to Angular concepts," Angular, [Võrgumaterjal]. Available: https://angular.io/guide/architecture. [Kasutatud 15 Oktoober 2020].

[87] „Introduction to services and dependency injection," Angular, [Võrgumaterjal]. Available: https://angular.io/guide/architecture-services. [Kasutatud 15 Oktoober 2020].

[88] „Workspace npm dependencies," Angular, [Võrgumaterjal]. Available: https://angular.io/guide/npm-packages. [Kasutatud 15 Oktoober 2020].

[89] „RxStomp," GitHub, [Võrgumaterjal]. Available: https://github.com/stomp-js/rx-stomp. [Kasutatud 17 Oktoober 2020].

[90] „Värvipsühholoogia," Hariduskeskus, [Võrgumaterjal]. Available: https://www.hariduskeskus.ee/opiobjektid/varvusopetus/vrvipshholoogia.html. [Kasutatud 17 Oktoober 2020].

[91] Bootstrap, [Võrgumaterjal]. Available: https://getbootstrap.com. [Kasutatud 17 Oktoober 2020].

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

[93] Dokumendipank, [Võrgumaterjal]. Available: http://www.dokumendipank.ee/sailitustahtajad/. [Kasutatud Oktoober 2020].

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

Mina, Sergei Pojev

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Paberivaba klienditeenindus pangas“, mille juhendaja on Meelis Antoi
   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.

05.01.2021

---

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

Lisa 2 – Serverirakenduse POM faili sisu

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.itcollege</groupId>
    <artifactId>paperfree</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>paperfree</name>
    <description>Paperfree Application</description>

    <properties>
        <java.version>15</java.version>
        <lombok.version>1.18.14</lombok.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-websocket</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
        </dependency>
        <dependency>
            <groupId>com.squareup.retrofit2</groupId>
            <artifactId>retrofit</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>com.squareup.retrofit2</groupId>
            <artifactId>converter-jackson</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>com.squareup.okhttp3</groupId>
            <artifactId>logging-interceptor</artifactId>
            <version>3.14.9</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
```

```xml
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Lisa 3 – Retrofit konfiguratsioon

```
package com.itcollege.paperfree.config;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.itcollege.paperfree.repository.SigningServiceInterface;
import okhttp3.OkHttpClient;
import okhttp3.logging.HttpLoggingInterceptor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import retrofit2.Retrofit;
import retrofit2.converter.jackson.JacksonConverterFactory;

@Configuration
public class RetrofitConfig {

    private final ObjectMapper objectMapper;

    @Value("${paperfree.signing.service.url}")
    private String signingServiceUrl;

    public RetrofitConfig(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    @Bean
    public SigningServiceInterface signingServiceInterface() {
        return this.retrofit().create(SigningServiceInterface.class);
    }

    @Bean("signingServiceRetrofit")
    public Retrofit retrofit() {
        HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
        interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
        OkHttpClient client = new OkHttpClient.Builder()
                .addInterceptor(interceptor)
                .build();

        return new Retrofit.Builder()
                .baseUrl(signingServiceUrl)

.addConverterFactory(JacksonConverterFactory.create(this.objectMapper))
                .client(client)
                .build();
    }
}
```

Lisa 4 – Serverrakenduse sõnumite saatmise teenuse implementatsioon.

```
package com.itcollege.paperfree.service;

public interface EventService {

    void processEvent(String type, String hostname) throws Exception;
    void processEvent(Event event) throws Exception;
}


package com.itcollege.paperfree.service.impl;

import com.itcollege.paperfree.service.Event;
import com.itcollege.paperfree.service.EventService;
import com.itcollege.paperfree.service.HandshakeService;
import com.itcollege.paperfree.service.TellerPc;
import jdk.jfr.EventFactory;
import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;

@Slf4j
@Service
public class EventServiceImpl implements EventService {

    private final SimpMessagingTemplate simpMessagingTemplate;
    private final HandshakeService handshakeService;

    public EventServiceImpl(SimpMessagingTemplate
simpMessagingTemplate,
                            HandshakeService handshakeService) {
        this.simpMessagingTemplate = simpMessagingTemplate;
        this.handshakeService = handshakeService;
    }

    private void processEvent(String type, TellerPc tellerPc, Object
content) throws Exception {
        if (StringUtils.isEmpty(type)) {
            throw new Exception("Type is missing.");
        }

        if (tellerPc == null) {
            throw new Exception("Teller PC information not
specified");
        }

        // We suppose, that correct implementation of handshakeService
is done.
        String clientPc = this.handshakeService
                .getClientPcHostname();

        if (StringUtils.isEmpty(clientPc)) {
            throw new Exception("Cannot find handshake for teller pc "
+ tellerPc.getHostname());
```

```java
        }

        var event = new Event(type);
        event.setDestination(clientPc);
        event.setContent(content);

        this.dispatchEvent(event);
    }

    @Override
    public void processEvent(String type, String hostname) throws
Exception {
        var tellerPc = new TellerPc();
        tellerPc.setHostname(hostname);

        this.processEvent(type, tellerPc, null);
    }

    @Override
    public void processEvent(Event event) throws Exception {
        if (event == null) {
            throw new Exception("Event object is null");
        }

        if (StringUtils.isEmpty(event.getDestination())) {
            throw new Exception("Destination not specified");
        }

        this.dispatchEvent(event);
    }

    @SneakyThrows
    private void dispatchEvent(Event event) {
        simpMessagingTemplate.convertAndSend(event.getDestination(),
event);
    }
}
```

```java
package com.itcollege.paperfree.service;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Event {

    private String destination;
    private String type;
    private Object content;

    public Event(String type) {
        this.type = type;
    }
}
```

```
package com.itcollege.paperfree.service;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class TellerPc {
    private Long id;
    private String hostname;
    private String activeUser;
}
```

Lisa 5 – Kliendirakenduse HTTP päringute baasteenus

```
import { Injectable } from '@angular/core';
import {HttpClient, HttpErrorResponse, HttpHeaders, HttpParams, HttpResponse}
from '@angular/common/http';
import {Observable, throwError} from 'rxjs';
import {NGXLogger} from 'ngx-logger';
import {mergeMap, map} from 'rxjs/operators';
import * as _ from 'lodash';

@Injectable({
  providedIn: 'root'
})
export class HttpService {

    constructor(private _httpClient: HttpClient,
                private _logger: NGXLogger) {
    }

    get<T>(endpoint: Observable<string>, headers: HttpHeaders, queryParams?:
object): Observable<T> {
        return endpoint
            .pipe(
                mergeMap((url: string) => {
                    const requestType = 'GET';
                    this._logRequest(requestType, url, queryParams);
                    return this._httpClient.get<T>(url, {
                        params: this._getQueryParams(queryParams),
                        headers: headers,
                        observe: 'response'
                    }).pipe(
                        map((response: HttpResponse<T>) =>
this._handleResponse(requestType, url, response)),
                    );
                }),
            );
    }

    post<T>(endpoint: Observable<string>, headers: HttpHeaders, body:
object): Observable<T> {
        return endpoint
            .pipe(
                mergeMap((url: string) => {
                    const requestType = 'POST';
                    this._logRequest(requestType, url, null, body);
                    return this._httpClient.post<T>(url, body, {
                        headers: headers,
                        observe: 'response'
                    }).pipe(
                        map((response: HttpResponse<T>) =>
this._handleResponse(requestType, url, response)),
                        );
                }),
            );
    }
    put<T>(endpoint: Observable<string>, headers: HttpHeaders, body: object):
Observable<T> {
```

```
        return endpoint
            .pipe(
                mergeMap((url: string) => {
                    const requestType = 'PUT';
                    this._logRequest(requestType, url, null, body);
                    return this._httpClient.put<T>(url, body, {
                        headers: headers,
                        observe: 'response'
                    }).pipe(
                  map((response: HttpResponse<T>) =>
this._handleResponse(requestType, url, response)),
              );
            }),
        );
    }

    delete<T>(endpoint: Observable<string>, headers: HttpHeaders,
queryParams?: object): Observable<T> {
        return endpoint
            .pipe(
                mergeMap((url: string) => {
                    const requestType = 'DELETE';
                    this._logRequest(requestType, url, queryParams);
                    return this._httpClient.delete<T>(url, {
                        params: this._getQueryParams(queryParams),
                        headers: headers,
                        observe: 'response'
                    }).pipe(
                        map((response: HttpResponse<T>) =>
this._handleResponse(requestType, url, response)),
                        );
                }),
                );
    }

    private _handleResponse<T>(requestType: string, url: string, response:
HttpResponse<T>): T {
        const data: T = response.body || {} as T;
        this._logResponse<T>(requestType, url, data);
        return data;
    }

    private _handleError(requestType: string, url: string, errorResponse:
HttpErrorResponse): Observable<never> {
        const errorMessage: string = errorResponse.message;
        this._logError(requestType, url, errorMessage);
        return throwError(errorMessage);
    }

    private _getQueryParams(inputParams: object): HttpParams {
    let params: HttpParams = new HttpParams();

    if (!_.isEmpty(inputParams)) {
        Object.keys(inputParams)
            .forEach((key: string) => {
                let value: any = inputParams[key];

                if (_.isNil(value)) {
                    return;
```

```
                }

                if (_.isPlainObject(value)) {
                    value = JSON.stringify(value);
                } else {
                    value = value.toString();
                }

                params = params.append(key, value);
            });
    }
    return params;
    }

    private _logRequest(requestType: string, url: string, queryParams?:
object, body?: object): void {
        const message: string = '\n' + requestType + ' request to ' + url +
'\n';

        if (!_.isEmpty(queryParams)) {
            this._logger.info(message, queryParams);
        } else if (!_.isEmpty(body)) {
            this._logger.info(message, body);
        } else {
            this._logger.info(message);
        }
    }

    private _logResponse<T>(requestType: string, url: string, data: T): void
{
        this._logger.info('\n' + requestType + ' response from ' + url +
'\n', data);
    }

    private _logError(requestType: string, url: string, errorMessage:
string): void {
        this._logger.error('\n' + requestType + ' error from ' + url + '\n',
errorMessage);
    }
}
```