TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Villem Maandi 155156 IAPB

PANGALINKIDE MAKSELIIDESE PROGRAMMEERIMINE

Bakalaureusetöö

Juhendaja: Gert Kanter
MSc

Tallinn 2020

Autorideklaratsioon

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

Autor: Villem Maandi

18.05.2020

Annotatsioon

Antud töö eesmärgiks oli luua eraldiseisev pangalinkide makseliides, mida on võimalik veebides lihtsasti kasutusele võtta. Uue makseteenuste komponendi loomine aitab vähendada pangalinkidega seonduvate muudatuste tegemiseks ja vigade parandamiseks kuluvat ressurssi.

Töö käigus valmis teek, mida on võimalik taaskasutada kõikides React raamistikel põhinevates projektides välise teegina. Lisaks loodi mikroveeb, mis võimaldab kasutada makseliidest ülejäänudes veebides. Teegi tarbeks loodi äriloogika kiht, mis pöördus kahe välise API poole pangalinkide andmete pärimiseks, kliendi suunamiseks pärast makse sooritamist, eeltäidetud maksekorralduse loomiseks ning tõlgete pärimiseks. Valminud teek võeti kasutusele olemasolevasse projekti, milles vana pangalinkide makselahendus asendati uue makseliidesega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 33 leheküljel, 6 peatükki, 22 joonist, 1 tabel.

Abstract

Interface Programming for Banklink Payments

The goal of this thesis is to create a user interface, which can be used to make payments in multiple websites in the company. The payments will be made using banklinks, which allow customers to pay bills or make other payments using their Estonian bank account. The payment component is a centralized way of maintaining banklink payment, which cuts down on resource usage for fixing issues and making changes considering banklinks for the company.

The project began by researching into a banklink payment component, which was used in one of the project in the company. The interface would have to be a reusable component, which could be used in both React and non-React websites. An empty repository was created in Atlassian Bitbucket. A backend project was created using Spring Boot framework. The usage of Billing API and Label API were required to make requests about payments and translations. A services gateway was used to authorize request in order to make those request. The work on a React banklink payment library began once the services had been created and tested. The template for the interface was partially reused for the project, but most of the frontend services had to be redeveloped. The library was uploaded to a private Artifactory NPM package repository, which is accessible to other React projects wanting to use this payment component. A microfrontend was also created for non-React websites, which displayed the library component. The values for the component would be read from the URL HTTP query parameters. The microfrontend was uploaded to a Kubernetes cluster, Atlassian Bamboo was used to build and deploy the website to a Docker container. The payment library and microfrontend were tested by creating unit tests and by individual testers in the company.

The thesis is in Estonian and contains 33 pages of text, 6 chapters, 22 figures, 1 table.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface*, rakendusliides
artifakt	Osa programmikoodist, mida on võimalik koodihoidlast jupina välja võtta
backend	Äriloogika, teenuste kiht
DNS	*Domain Name Server*, teisendab IP aadressid domeeninimedeks
endpoint	Aadress, mida kasutatakse veebiteenusega suhtlemiseks
epic	JIRA ülesande tüüp, mille alla võib kuuluda mitu erinevat tööd
HTTP	Teabe edastamise protokoll arvutivõrkudes
JSON	*JavaScript Object Notation*, teksti kujul lihtsustatud andmevahetusvorming
klaster	Kogum süsteemidest, mis töötavad ühtse tervikuna
konteiner	Teostab operatsioonisüsteemi tasemel virtualiseerimist [1]
küpsis	Tekstikujuline andmeplokk kliendi veebibrauseris [2]
mikroveeb	Eraldiseisev veebileht, mida on võimalik implementeerida olemasolevasse veebilehte
pod	Grupp ühest või mitmest konteinerist jagatud võrgu ja spetsifikatsiooniga
proksi	Arvutivõrgu server, mis vahendab infovahetust kliendi ja serveri vahel [3]
pull request	Toiming, mille jooksul teised arendajad teevad koodi ülevaatust, jälgides koodi kvaliteeti ning koodis olevaid vigu
REST	*Representational State Transfer*, tarkvaraarhitektuuri stiil
SGW	*Services Gateway*, teenuste proksi
story	Atlassian JIRA keskkonnas olev ülesanne, mis peab läbima töövoo, osa agiilsest arendusest
token	Räsiväärtus, mille järgi on võimalik tuletada algne väärtus
toru	„Toru ehk konveier (inglise keeles *pipeline*) on kogum jadamisi ühendatud andmete töötlemise elemente, kus ühe elemendi väljund on järgmise elemendi sisendiks.“ [4]

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

1.1 Taust

Bakalaureusetöö aluseks on projekt ettevõttes, mille eesmärk on luua uus makseteenuste komponent, mis võimaldab teha makseid pangalinkide kaudu. Makseliidest hakkavad kasutama ettevõttesse kuuluvad kliendid. Pangalink võimaldab sooritada makset eeltäidetud maksekorraldusega, mis teeb kliendi jaoks maksmise lihtsamaks. Pangalingi maksete komponent on eelkõige mõeldud klientidele arvete tasumise eesmärgil, kuid sellel on ka mitmeid muid kasutuskohti. Näiteks on võimalik komponendi kaudu teha ettemakseid või tasuda teenuste eest.

Uus lahendus põhineb olemasoleval makseteenuste komponendil, mis on ettevõtte iseteeninduse projektis varasemalt kasutusel. Töö eesmärk on võtta aluseks projektis olev makseteenuste lahendus ning teha sellest taaskasutatav komponent, mida on võimalik implementeerida kõikides ettevõtte projektides, mis puutuvad kokku maksete tegemisega. Uus lahendus muudab selliste projektide haldamise oluliselt lihtsamaks.

1.2 Lahendatav probleem

Ettevõttes on pangalingid kasutusel 14 erinevas projektis, kusjuures igas projektis on pangalinkide makseliides individuaalselt implementeeritud. Probleemi lahendamiseks loodi uus eraldiseisev komponent, mida on igas projektis võimalik lihtsasti kasutusele võtta. Uue lahendusega muutuvad pangalingimaksed tsentraalseks süsteemiks, mille haldamine on lihtsam võrreldes varasema lahendusega. Lisaks väheneb pangalinkide muutmisele ning vigade otsimisele kuluv ettevõtte ressurss.

1.3 Metoodika

Projekti arendus toimus individuaalselt. Vajadusel sai arendusse kaasata kogenenum arendaja, et saada juhiseid keerulisemate probleemide lahendamiseks. Projekti ärilised nõuded pani paika ettevõtte arvelduse osakonna süsteemiarhitekt ning arendustiimis

pandi paika projekti valmimise ajaline hinnang ning valmislahenduse nõuded. Kogu arendustöö toimus operatsioonisüsteemis Microsoft Windows 10 kasutades integreeritud arenduskeskkonda IntelliJ IDEA [5]. Versioonihalduseks oli kasutuses Git [6]. Presentatsioonikiht on programmeeritud kasutades React [7] JavaScript raamistikku, mis suhtleb Spring Boot [8] raamistikul põhineva äriloogika kihiga.

1.4 Töö ülesehitus

Töö sisus antakse esmalt ülevaade projekti arhitektuurist. Seejärel põhjendatakse valitud raamistike ning teekide valikuid, mida kasutati liidese programmeerimiseks. Edasi järgneb projekti järkjärguline realisatsioon. Kokkuvõttes antakse hinnang valminud tööle.

Loodud lahenduse programmikood asub ettevõtte privaatses Bitbucket [9] salves.

2 Analüüs

Selles peatükis tuuakse välja veebirakenduse funktsionaalsed ja mittefunktsionaalsed nõuded, rakenduse arhitektuurivaated ning selgitatakse töö planeeringut.

2.1 Süsteemi üldine kirjeldus

Programmeeritakse lahendus, mis võimaldab klientidel pangalinkide kaudu ettevõtte arvelduskontodele makseid teha. Valminud lahendus peab vastama sätestatud funktsionaalsetele ning mittefunktsionaalsetele nõuetele. Lisaks peab lahendus käima kokku süsteemi arhitektuuriga.

Lahendus peab töötama kolmes keskkonnas - *development*, *test* ja *live*. Kliendile on ligipääsetav ainult *live* keskkond. Enne rakenduse *live* keskkonda paigaldamist tuleb lahendust testida. *Development* ja *test* keskkonnad on loodud arendajatele ning testijatele lahenduse toimimise kontrollimiseks ning testimiseks.

2.2 Funktsionaalsed nõuded

Funktsionaalseid nõudeid vaadeldakse ettevõtte kliendi seisukohast.

- Nii anonüümne kui ka isikustatud klient saab pangalingi kaudu maksta mistahes summaga Swedbank, SEB, Danske, Luminor, LHV ja Coop panka
- Pangalingist tagasitulekul peab kliendile olema vastav teade makse õnnestumise või tühistamise kohta

2.3 Mittefunktsionaalsed nõuded

Mittefunktsionaalsed nõuded määravad ära makseteenuse tehnilise poole.

- Rakendus peab olema arendatud React mikroveebina, mille funktsionaalsust saab taaskasutada nii React kui ka mitte-React keskkondades

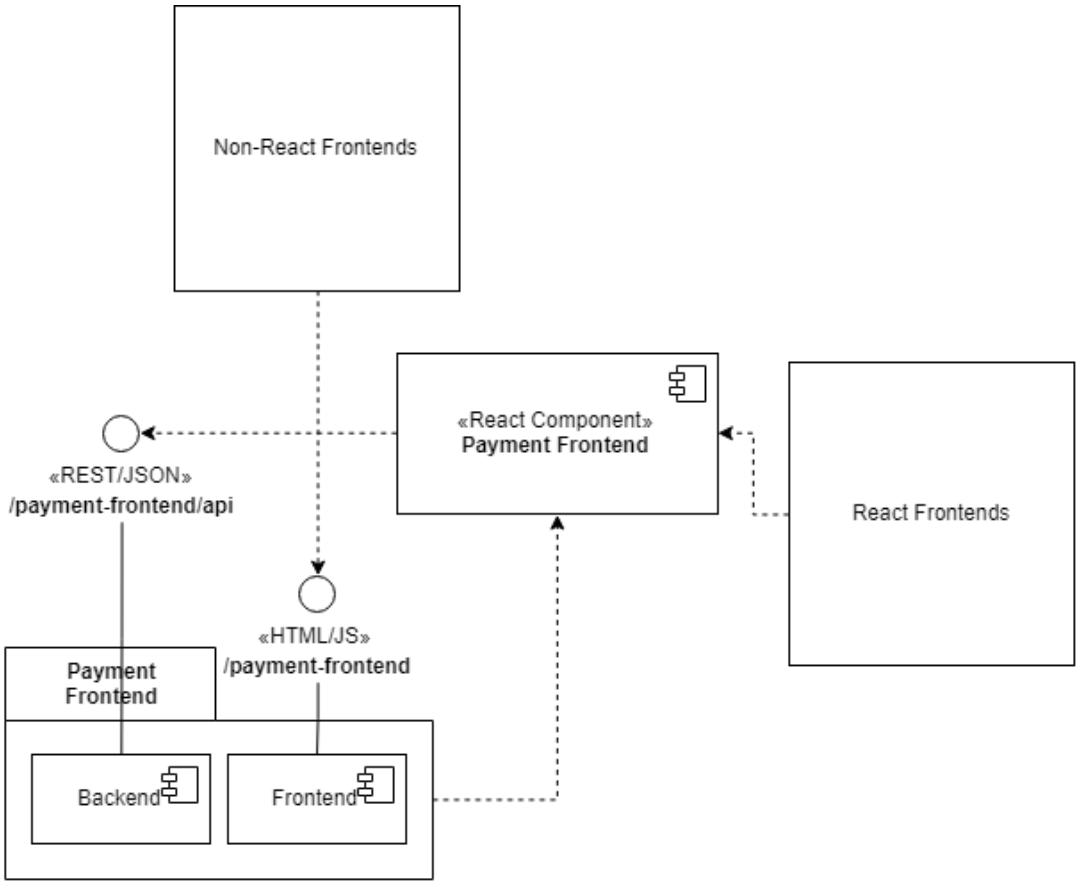
- Reacti puhul peab olema tegemist teegiga, mida saab React projektides NPM-i [10] (*Node Package Manager*) kaudu paigaldada
- Teek peab olema kättesaadav ettevõtte privaatsest Artifactory [11] NPM registrist
- Erinevate keskkondade tugi peab olema teeki parameetrina toetatud ning vastavad keskkonna URL-id peavad olema komponenti sisse kirjutatud
- Teek peab olema mitte-React keskkondades lihtsasti kasutatav veebilehe sees vastava keskkonnda URL-i pealt, parameetrid peavad tulema kaasa HTTP päringusõnedena
- Lahendus peab olema paigaldatud standardse Kubernetese [12] mikroveebina *vkube* klastrisse

2.4 Arhitektuur

Järgnevalt on kirjeldatud rakenduse arhitektuuri, millele süsteem peab vastama. Välja on toodud nii rakenduse vaated kui ka arenduse ja paigalduse vaated.

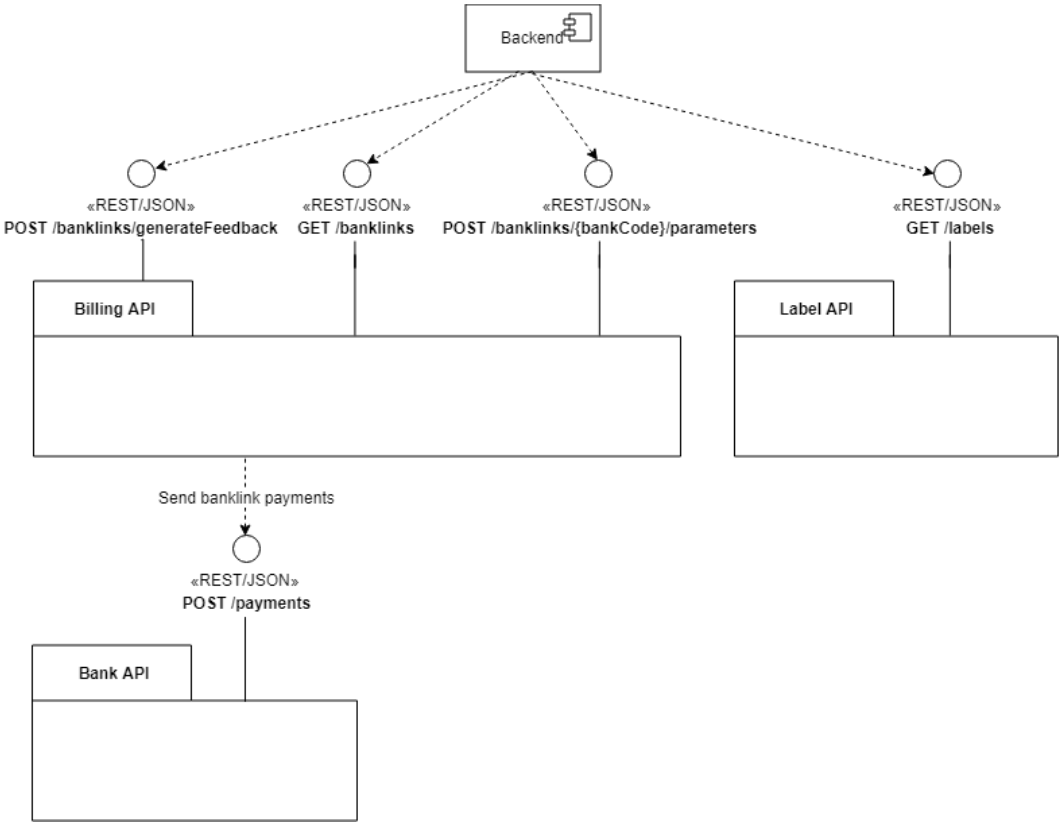
2.4.1 Rakenduse mudelid

Järgnevalt kujutatakse makseteenuste liidese implementeerimist nii Reacti-põhistes kui ka mitte-React raamistikul üles ehitatud veebides (Joonis 1). Mitte-React keskkondade puhul luuakse eraldi veebileht, kus kuvatakse teegis olevat komponenti. React raamistikel põhinevates rakendustes on võimalik teeki kasutada ilma veebileheta. Reacti komponent pöördub makseteenuste liidese API rakenduse pihta.



Joonis 1. Rakenduse arhitektuurivaade.

Liideses peab olema võimaldatud teha päringuid Billing API ja Label API teenuste pihta (Joonis 2). Billing API kaudu päritakse andmeid pangalinkide kohta, mille järgi koostatakse pangalinginupud. Lisaks kasutatakse teenust pankadega suhtlemiseks, mille korral valmistatakse andmed ette maksekorralduse andmete määramiseks. Billing API poole pöördutakse samuti, kui panga kaudu saadakse vastus makse õnnestumise või ebaõnnestumise kohta. Klient suunatakse lehele, kus on vastav teavitus makse kohta. Kasutajaliidese tõlked päritakse Label API teenuse kaudu, kus päritakse nii eesti- kui ka venekeelsed tõlked.



Joonis 2. Rakenduse päringud väliste API-de teenuste pihta.

2.4.2 Rakenduse arenduse ja paigalduse mudelid

Rakenduse arenduse ja paigalduse vaates on kujutatud rakenduse paigaldamist Kubernetesesse ja OpenStack Swifti [13] (Lisa 1). *Vkube* klaster on mõeldud välisteenuste jaoks, millele on ligipääs otse internetti avatud. Kubernetese klastrid on loodud *active-active* teenuste tarbeks, mille korral kaks *node* ehk serverit sünkroniseerivad omavahel andmeid (inglise keeles *synchronous mirroring*). Kubernetese *vkube* klastris asuvad kaks eraldi töökoormusega serverit. Kui põhiserver ei tööta, siis võtab lisaserver üle põhiserverilt kõik seadmed, kliendi ühendused ja muu kuniks põhiserver taas tööle hakkab [14]. Klastrite avalik ligipääs käib üle globaalse DNS-i [15] (*Domain Name Server*), mis jagab koormuse erinevate õlgade vahel laiali.

Kubernetese klastrite vaates on kujutatud *vkube* klastri kahe serveri (*adr-vkube* ja *mus-vkube*) tööpõhimõtet (Lisa 2). Nginx on proksi, mida kasutatakse äriloogika kihi teenuste ja presentatsioonikihi staatiliste ressursside pärimiseks. OpenStack Swifti sisu serveeritakse Nginx [16] proksi kaudu.

2.5 Töö planeerimine

Töö käigus kasutati agiilse tarkvaraarenduse meetodeid. Kuna projekt oli mahukas, siis jaotati selle arendus järgmisteks väiksemateks töödeks. Iga töö kohta koostati Atlassian JIRA [17] keskkonnas uus *story* ehk ülesanne. Mõne ülesandega sai tegeleda samaaegselt, kuna need ei sõltunud teineteise valmisolekust. *Storyde* hulka kuulusid järgmised tööd:

- äriloogika kihi ehk teenuste programmeerimine
- presentatsioonikihi ehk mikroveebi programmeerimine
- pideva integreerimise ja tarne ülesseadmine
- React teegi lisamine Artifactory NPM registrisse
- rakenduse paigaldamise seadistused Kubernetese keskkonna jaoks
- uue makseteenuste komponendi kasutusele võtmine ettevõtte iseteeninduses
- komponendi tehnilise dokumentatsiooni loomine

Projekti arendusega tegeleti iganädalaselt. Osaliselt rakendati paarisprogrammeerimist, mis andis võimaluse sujuvama ning kiirema arenduse toimiseks, samuti oli suurem tõenäosus, et arenduse käigus tekib vähem vigu. Arenduse käigus oli oluline luua üksuste teste, mis pidid paki ehituse ajal edukalt läbima. Töö raames tegid teised arendajad koodiülevaatusi tehtud arendustele.

3 Tehnoloogiad

Järgnevates alampeatükkides tutvustatakse projekti arendamise käigus kasutatud tehnoloogiaid. Esmalt kirjeldatakse pangalingi tööpõhimõtet ning spetsifikatsioone, seejärel arenduse käigus kasutatud keskkondi ja tööriistu. Samuti põhjendatakse teekide ning raamistike valikuid nii äriloogika kihi kui ka presentatsioonikihi puhul.

3.1 Pangalink ja eeltäidetud maksekorraldus

Pangalink võimaldab makset sooritada otse veebilehelt. Veebirakendus saadab pangale eeltäidetud maksekorralduse ning pank saadab rakendusele vastuse makse sooritamise, ebaõnnestumise või katkestamise kohta. Pangalingi kaudu on võimalik ka kasutajat autentida ehk kontrollida kasutaja identiteeti.

Kõik Eesti pangaliitu kuuluvad pangad kasutavad alates 2014. aasta oktoobrist sama tehnilist lahendust, et muuta makse- ning autentimislahendused turvalisemaks [18]. Eesti pangad toetavad sama tehnilist spetsifikatsiooni. Pangalingi spetsifikatsioon [19] kirjeldab päringu teenuseid, mida on pangalinkide korral võimalik kasutada. Antud töö käigus kasutatakse teenust 1011, mille käigus saadetakse panka allkirjastatud maksekorralduse andmed, mis ei võimalda kliendil enam internetipangas andmeid muuta. Lisaks kasutatakse teenust 1111, mille puhul tuleb kaasa saata maksekorralduse andmed. Andmete hulgas peavad olema määratud näiteks päringu algatamise kuupäev, saaja konto number ja maksmisele kuuluv summa. Eduka makse sooritamisel koostatakse kaupmehele teenuse 1111 päringu vastus, mida kasutatakse eestisisese maksekorralduse toimumise teavitamiseks. Ebaõnnestunud makse puhul tagastatakse teenuse 1911 vastus.

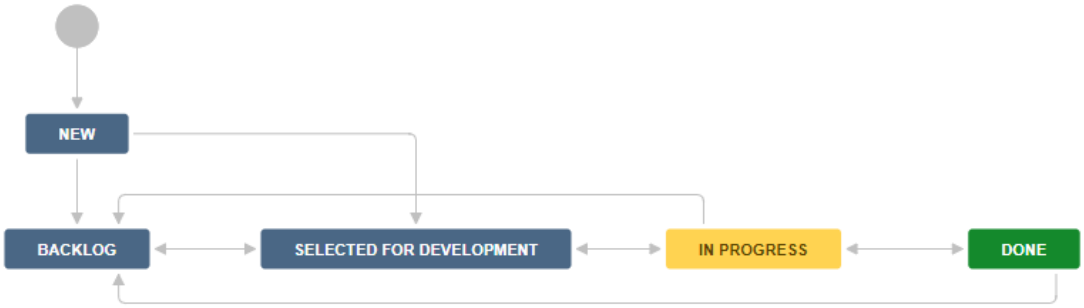
3.2 Projekti keskkonnad ja tööriistad

Projekti arenduse käigus kasutati mitmeid keskkondi ja tööriistu, mis hõlbustasid töö kiiremat valmimist. Järgnevalt on kirjeldatud keskkondi ning nende seoseid antud tööga.

3.2.1 Atlassian JIRA

Atlassian JIRA on platvorm, mis on mõeldud projekti agiilseks arenduseks. JIRA platvorm võimaldab luua uusi töid, mis on seotud projekti arendusega. Tööde hulka võivad kuuluda uued arendustööd, vigade parandustööd või muud tööd, mis ei vaja programmikoodi kirjutamist. JIRA võimaldab loodud töid kergesti hallata. Kuna kõik tööd läbivad töövoo (Joonis 3), siis on lihtne jälgida tööde hetkeseisu. Töövoo kirjeldusi genereeritakse automaatselt JIRA keskkonnas *story* kõikide võimalike staatuste põhjal.

Kogu pangalinkide projekti jaoks koostati *epic* tüüpi töö, mille alla kuulusid osadeks jaotatud tööd ehk *storyd*, mis seati paika töö planeerimise alguses. *Epic* tüüpi töö kirjelduses on välja toodud nii kogu projekti lühikirjeldus kui ka funktsionaalsed ja mittefunktsionaalsed nõuded. Iga *story* juures on lühike kirjeldus töö nõuete kohta, mis aitab töö sisust paremini aru saada.



Joonis 3. JIRA keskkonna ühe töö näidisvoog.

3.2.2 Atlassian Bitbucket

Projekti arenduse käigus kasutati Atlassian Bitbucket haldustarkvara, mis võimaldas luua uusi repositooriume ehk salvi. Bitbucket sarnaneb GitHub koodihoidlaga. Bitbucketil on võimalus ühenduda teiste Atlassiani keskkondadega nagu JIRA või Bamboo [20].

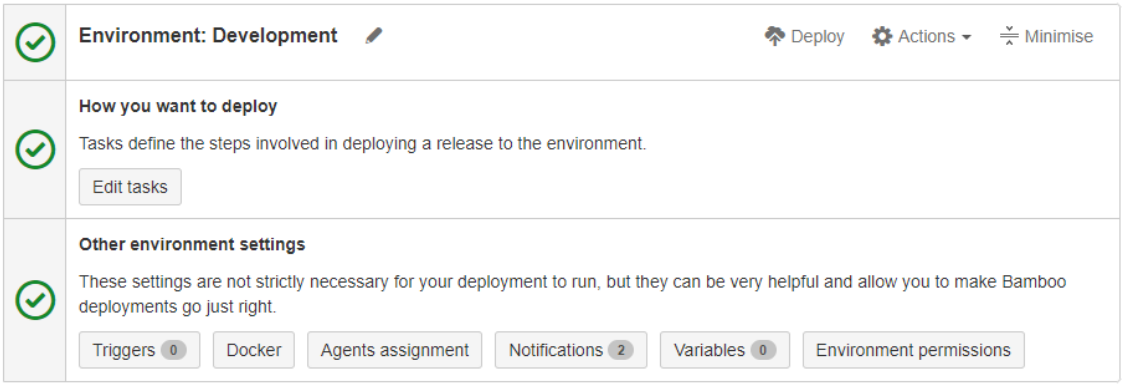
Uue salve loomiseks Bitbucketis peab eksisteerima projekt, kuhu seda lisada. Ettevõttes oli olemas arvelduse projekt, kuhu uut salve lisada. Salve nimeks sai payment-frontend, mis väljendab selgelt ja lühidalt projekti sisu. Rakenduse äriloogika kiht, rakenduse teek ja mikroveeb asuvad ühes salves, kuid asetsevad eraldi kaustades.

3.2.3 Atlassian Bamboo

Bamboo on Atlassiani poolt välja arendatud CI *(Continuous integration)* / CD *(Continuous delivery)* [21] server, mida kasutatakse pakkide ehitamise ning paigaldamise vahendina. CI on praktika, kus tiimi erinevad liikmed integreerivad pidevalt kirjutatud koodi. Pärast koodis muudatuste tegemist ehitatakse rakenduse uus versioon keskses CI serveris kokku ja käivitatakse automaattestid, mis kontrollivad, et kõik töötab jätkuvalt ootuspäraselt. CD on praktika, kus tarkvara arendatakse nii, et seda oleks võimalik *live* keskkonda paigaldada igal ajahetkel. CD on CI jätk ja puudutab rakenduse paigaldamist erinevatesse keskkondadesse.

Bamboos on nähtavad kõik *build* projektid ja *planid*. Selleks, et Bamboo integratsiooni kasutada, peab Bamboo *build plan* olema seotud Bitbucketi koodihoidlaga. *Builde* jooksutatakse üldjuhul *remote agentide* pealt, mis asuvad ühel masinal. Masinasse on paigaldatud enim vajaminevad pakette ja Java versioone, mis rahuldavad enamike *build planide* vajadusi.

Deploy jaoks kasutatakse Bamboos *deployment* projekte, mille alla on seadistatud keskkonnad, kuhu projekti paigaldadakse (Joonis 4). *Deploysid* jooksutatakse masina pealt, kuhu on paigaldatud mitu agenti.



Joonis 4. Bamboo *development* keskkonna paigalduse vaade.

3.2.4 Kubernetes

Kubernetes (lühendatult K8s) on konteinerite haldussüsteem, mille abil saab mugavalt jooksutada Dockeri konteinereid klasterdatult ja mitmed veebisaidis. Ettevõttes kasutatakse välis- ja siseteenuste tarbeks eraldi klastreid. Ettevõttes luuakse iga

valdkonna jaoks eraldi nimeruumid, mis grupeerivad *pode*, teenuseid jne. *Pod* on grupp ühest või mitmest konteinerist. See võimaldab teenuseid ka üksteisest loogiliselt eraldada [22]. Kõik paroolid ja konfiguratsioonifailid, mis on antud projektis kasutusel, on hoiustatud projektisiseselt. Autentimine toimub kasutajanime ja *tokeni* alusel.

Rakenduse toru ehk konveierina on kasutatud Bitbucketit, Bambood ja Kubernetest. Bitbucketi salves on lisaks programmikoodile ka Dockeri ja Kuberentese seadistused ja skriptifailid. Salv on ühenduses Bambooga ja peale *master* harusse koodi üleslaadimist toimub automaatselt testide käivitamine, Dockeri *image* ehitamine ning *image* üleslaadimine Artifactory registrisse. Bamboost käivitatakse antud paki *deploy* ehk paigaldamine kas käsitsi või automaatselt *development*, *test* või *live* keskkonda.

3.2.5 Docker

Dockerit võib võrrelda virtuaalmasinaga, mis vajab käivitamiseks oluliselt vähem ressursse. Näiteks virtuaalmasina käivitamine võtab aega mitmeid minuteid, sest iga virtuaalmasina käivitamiseks peab selle sees olev operatsioonisüsteem ennast üles laadima. Konteinerid isoleerivad ja kapseldavad rakenduste tööd *host* masinas.

Dockeri puhul saab lihtsasti ja standardselt rakendusi installeerida. Konteinerit võib kujutada kui operatsioonisüsteemi teise operatsioonisüsteemi sees, millesse saab installeerida ja käivitada rakendusi ning konteinerid sarnaselt virtuaalmasinaga. Konteinerite lahendus ei ole midagi uut, kuna Google on seda kasutanud edukalt üle 10 aasta [23]. Konteinerite kasutus aitab muuta osa arhitektuurist, mis puudutab erinevate rakenduste jooksutamist ja skaleeritavust.

3.2.6 Artifactory

Artifactory on universaalne pakihaldur, mis toetab erinevaid pakihalduse formaate nagu Gradle [24], NPM ja Docker. Lokaalsed salved asuvad ettevõtte sisevõrgus ning talletavad ainult ettevõtte ehitatud pakke. Artifactory salves hoitakse makseliidese teeki, mis ehitatakse kokku ning laetakse üles Bamboo keskkonnas NPM-i kaudu. Samuti on Artifactorys Docker failide salv, kuhu laetakse *backend* rakenduse *build* failid automaatselt pärast edukat Bamboo *buildi*.

3.2.7 OpenStack Swift

OpenStack Swift on jagatud objekti hoiustamise süsteem (inglise keeles *multi-tenant object storage system*), mis on mõeldud üle REST HTTP API struktureerimata andmete talletamiseks. Swift tegeleb loodud mikroveebi ning *backendi* ehk äriloogika kihi paigaldamisega konteinerisse. Kui konteinerit ei eksisteeri, siis luuakse uus. Antud projektis loodud makseliides hakkab olema kättesaadav välisteenuste tarbeks, seega paigaldatakse projekt konteineris olevasse välisteenuste ehk *vkube* klastrisse. *Vkube* klastris olevad teenused on internetis avalikud.

3.2.8 Services Gateway

Services Gateway (SGW) on teenuste proksi, mis teostab kahte olulist tegevust. Üheks tegevuseks on kasutaja autentimine ja teenuste kasutamise autoriseerimine ning teiseks teenuste lõpp-punktide (inglise keeles *endpoint*) abstraheerimine.

Kuna rakendustel ei ole lubatud otse API-de teenustele päringuid teha, siis kasutatakse rakenduses SGW proksit. Antud rakenduses loodi eri keskkondade tarbeks kolm rakendusepõhist SGW kasutajat. Kasutajatele tuli päringute tegemiseks anda õigused *endpointi* ehk veebiteenusega suhtlemise aadressi põhiselt, kuna rakendus pöördub kahe välise API poole. Makseteenuste rakenduses tehakse väliste teenuste pihta nii GET kui ka POST päringuid.

Label API puhul kasutatakse ühte *endpointi*:

- GET /rest/service/label-api/labels

Billing API puhul kasutatakse kolme erinevat *endpointi*:

- GET /rest/Billing/billing-api/banklinks
- POST /rest/Billing/billing-api/generateFeedback
- POST /rest/Billing/billing-api/{bankCode}/parameters

3.2.9 Nginx

Nginx paigaldatakse koos äriloogika kihiga igasse klastrisse. Nginxi kasutatakse proksina, et teha presentatsioonikihist päringuid äriloogika kihi teenuste ning presentatsioonikihi staatiliste ressursside pihta. Proksi kuulab pealt aadressil localhost:80.

Rakenduses tehakse suunamisi kindlatele *pathidele* ehk asukohtadele, selleks et rakendus saaks vajalikud ressursid kätte (Tabel 1).

Tabel 1. Nginx suunamiste konfiguratsioon.

Asukoht	**Proksi**	**Ülekirjutamine**
/ (root)	https://s3-vsite.estpak.ee	/frontend-bundles/build/index.html
/payment-frontend/js/	https://s3-vsite.estpak.ee	/frontend-bundles/payment-frontend/js/
/payment-frontend/fonts/	https://s3-vsite.estpak.ee	/frontend-bundles/payment-frontend/fonts/
/payment-frontend/css/	https://s3-vsite.estpak.ee	/frontend-bundles/payment-frontend/css/
/payment-frontend/svg/	https://s3-vsite.estpak.ee	/frontend-bundles/payment-frontend/svg/
/payment-frontend/api/	http://payment-frontend-api:8080/payment-frontend/api/	-
/health	-	-
/heartbeat	-	-
/static-404	-	-

3.3 Raamistike ja teekide valikud

Järgnevalt on ära kirjeldatud presentatsioonikihis ning äriloogika kihis kasutatud raamistikud ja teegid.

3.3.1 React

Suures osas ettevõttes olevates projektides on presentatsioonikihi üles ehitatud React JavaScript raamistikul, mille on arendanud Facebook. Reacti kasutatakse veebirakendustes, kus andmed muutuvad ajas ning samuti ei vaja veebirakendus uuenduste tegemiseks kogu lehe täielikku laadimist. Reactiga tagatakse rakenduse kiirus, skaleeritavus ja lihtsus. Reacti alternatiivideks on populaarsemate seast näiteks Angular 2 või VueJS. React rakendusele on võimalik lisada sõltuvusi ehk teisi pakette NPM registrite kaudu.

Projektis on rakendatud tüübipõhist koodikirjutamist kasutades TypeScript [25] programmeerimiskeelt, mida arendab Microsoft. TypeScriptis kirjutatud kood kompileerub tavaliseks JavaScriptiks. Tüübikirjeldus annoteerib muutujate tüüpe sarnaselt programmeerimiskeelele Java ning tüübikontroll viiakse läbi kompileerimise ajal.

Webpack [26] on JavaScripti moodulite kokkupakkija. Webpacki kasutatakse projektis presentatsioonikihi ehitamisel. Webpack kompileerib kogu projekti .html .css ja .js faililaiendustega failideks, mida on võimalik ka veebi otse üles laadida ning veebibrauserites avada. Webpackile on võimalik luua eraldi konfiguratsioonifail, mis loetakse sisse Webpack käskude jooksutamisel.

MobX [27] on lihtne ja skaleeritav *state-management* teek, mida kasutatakse rakenduses erinevate väärtuste haldamiseks. MobX abil määratakse rakenduse komponentide seisundeid. Seisundites võib hoida erinevat tüüpi muutujaid, mida on võimalik rakenduse töö käigus lugeda ja muuta. MobX tagab vastavad väärtused ja React renderdab uuendatud komponente, milles kuvatakse uusimaid andmeid.

Axios [28] on HTTP klient, mille abil on võimalik teha päringuid äriloogika kihi teenuste pihta. Antud projektis tehakse GET ja POST päringuid, et pärida rakenduse tõlkeid, saada infot olemasolevate pangalinkide kohta, kliendi panka suunamiseks ning kliendi õigele URL-ile suunamiseks.

Kasutajaliidese komponentide kujundus on loodud ettevõtte UI/UX (*user interface/user experience*) tiimi poolt, kes vastutab komponentide loomise eest, mida on võimalik React raamistukul põhinevates projektides kasutada. UI komponentide hulgas on nuppe, tekstivälju ja muid kujundatud komponente, mida on lihtne implementeerida ning mille kasutamine tagab ettevõtte brändi stiili kõikides veebirakendustes.

3.3.2 Spring Boot

Liidese äriline kiht on üles ehitatud Spring Boot raamistikul, mis on enamlevinud raamistik Spring [29] rakenduste loomisel. Spring Boot võimaldab arendajatel keskenduda koodi kirjutamisele, kuna enamik seadistusi ning teekide allalaadimisi teeb raamistik ise. Samas on raamistik arendajatele samaväärselt seadistatav nagu Spring.

Antud töö puhul kasutati Spring Boot versiooni 2.0.2.RELEASE ning rakenduse ehituse tööriistana kasutati Gradle versiooni 5.3.1. Gradle eelis võrreldes samalaadse tööriista Maven-iga on konfiguratsioonifailide kompaktsus. Rakenduse jooksutamisel kasutati Java versiooni 8.

Lisaks kasutatati projektis keskset sessioonihoidlat Redis [30], mis võimaldab pangast tulnud tagasiside korral kliendi *tokeni* järgi õigele lehele suunata. Kuna pangalingid võimaldavad määrata URL-i ainult õnnestunud makse ja ebaõnnestunud makse korral, siis kasutati Redise andmebaasi, et oleks võimalik kliente suunata rohkematele lehtedele. Redises luuakse *token*, mille järgi on hiljem võimalik lugeda kliendi sessioonist välja URL-id, kuhu klienti suunata juhtudel, kui makse õnnestus, ebaõnnestus või makset veel töödeldakse ehk ettevõtte süsteemidesse pole makse veel kohale jõudnud.

4 Rakenduste programmeerimine

Ettevõttes on pangalinke kasutatud erinevates projektides. Töö alustuseks uuriti ühte ettevõtte projekti, kuhu oli makseteenuste lahendus varasemalt arendatud. Töö aluseks sai valitud projekt, milles olev pangalinkide makseteenus oli tõestanud oma töökindlust. Ettevõtte iseteenindus sobis projekti aluseks. Uuriti milliseid komponente saaks võimalusel taaskasutada ja mis vajaksid uue lahenduse puhul ümberarendusi. Selle põhjal sai alustada eraldiseisva pangalinkide makseliidese arendamist.

4.1 Makseliidese backend rakendus

Esmalt tuli Atlassian Bitbucket keskkonnas luua uus *feature* haru olemasolevasse payment-frontend salve. *Feature* haru on üldjuhul seotud JIRA *taskiga*. Iga arenduse jaoks tuli luua eraldi haru *master* haru pealt. *Master* harusse ehk põhiharusse ei ole programmeerijatel üldjuhul õigusi oma koodi otse üles laadida, kuna *master* haru põhjal paigaldatakse erinevatesse keskkondadesse rakenduse pakke ning arendused peavad ennem olema kontrollitud. Pärast edukat koodiülevaatust on programmeerijal õigus oma kood *feature* harust *master* harusse üles laadida.

Pangalinkide makseteenuse liideses kasutatakse Spring Boot rakendust. Projekti põhja loomiseks on kasutusel Spring Initializr [31] tööriist, mis tekitab rakenduse põhja koos esmaste paketisõltuvustega. Raamistiku põhi ja kõik muu programmikood asub arenduse *feature* harus. Gradle võimaldab IntelliJ IDEA-s automaatselt sõltuvusi alla laadida, mis asuvad *build.gradle* failis. IDEA-s on Gradle tööriistad sisseehitatud, mis võimaldavad käivitada erinevaid käske ilma käsurida kasutamata.

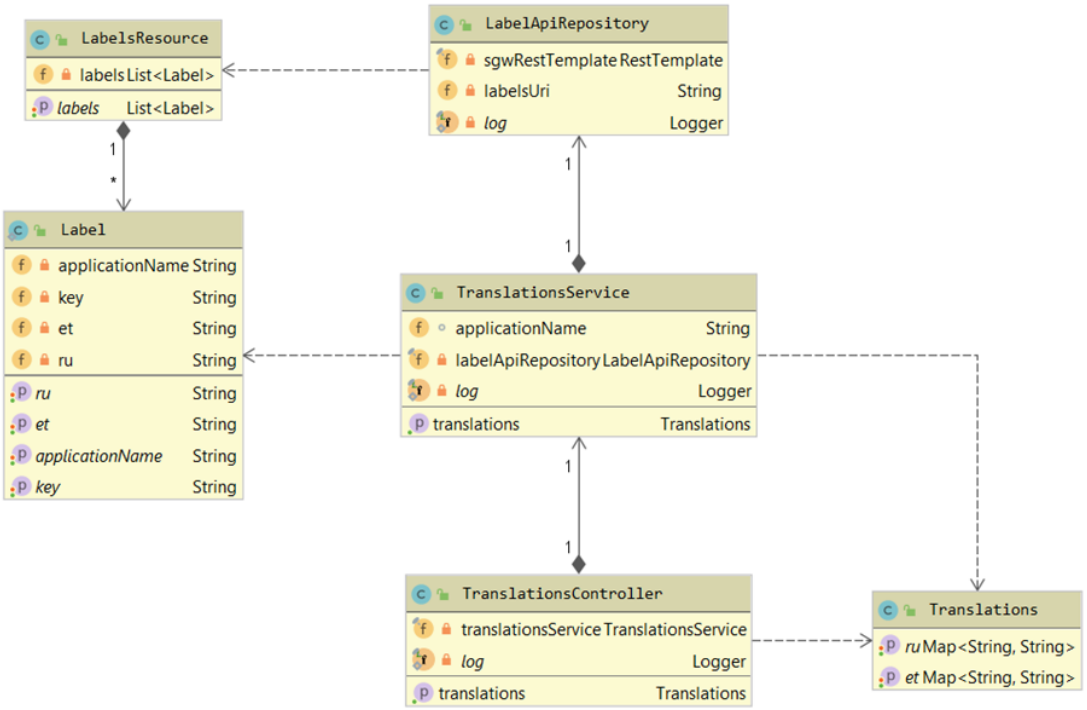
Projekti *backendis* kasutatakse *controller-service-repository* arhitektuuri, mis koosneb kolmest kihist. *Controlleris* ehk esituskihis on teenused, mida makseliides kasutab. Andmete valideerimine ja päringud tehakse *service* ehk äriloogika kihis. *Repository* ehk andmepääsu kihis päritakse andmeid API-de teenuste kaudu. Antud rakendus kasutab väliseid API teenuseid, mis pärivad andmeid erinevatest andmebaasidest. Äriloogika kiht on vahenduslüli esituskihi ja andmepääsu kihi vahel. Äriloogika kiht töötleb saadud

andmeid ja muudab need vahenduslüli jaoks sobivaks [32]. Üks teenusloogika klass võib kasutada mitut andmepääsu kihti. Äriloogika kihid võivad olla omavahel ühenduses.

Rakenduse andmepääsu kihis pöördutakse Billing API ja Label API poole. Nimetatud väliste API rakenduste pihta päringute tegemiseks pidid olema rakendusel SGW kasutajad, mida kasutatakse erinevates keskkondades päringute tegemisel autentimismeetodina. SGW seadistused on tehtud *application.yml* failides. Iga keskkonna jaoks on eraldiseisev seadistuste fail, kus asuvad kasutajanimed, paroolid ning muud väärtused, mille väärtusi rakendatakse vastavalt keskkonnale. Rakendus laetakse üles kolme erinevasse keskkonda. Igal kasutajal on õigus teha päringuid vastavas keskkonnas spetsiifiliste API *endpointide* pihta.

4.1.1 Tõlgete pärimise teenus

Rakendus peab olema kasutatav nii eesti kui ka vene keeles. Rakenduses on loodud tõlgete pärimise teenus. Teenus tagastab rakenduses kasutatavate tekstide tõlked, et veebipoolsed rakendused saaksid kuvada õigeid tekste olenevalt veebis kasutatavale keelsusele. Tõlkeid päritakse Label API rakenduse teenuse kaudu. Teenuse lahendus on kujutatud klassiskeemil (Joonis 7).



Joonis 5. Tõlgete pärimise teenus klassiskeemil kujutatuna.

Äriloogika kiht pöördub Label API *endpointi* poole, milleks on /rest/service/label-api/labels, mille kaudu päritakse rakenduse tõlked. Label API suhtleb tõlgete keskhoidlaga, mis võimaldab rakendusepõhiselt tõlkeid lisada, muuta ning vajadusel ka kustutada. Koostöös ettevõtte Label API süsteemihaldajaga loodi tõlgete keskhoidlasse uus rakenduse, mille nimeks sai paymentweb. Label Central on ettevõttesisene veebipõhine tõlgete haldusliides, mis võimaldas lisada iga keskkonna jaoks nii eesti- kui ka venekeelseid tõlkeid. *Endpointi* pihta päringute tegemisel oli oluline ära määrata *application-name* parameeter, mille väärtuseks on rakenduse enda nimi.

Payment Frontend *backend* rakenduses on *TranslationsController* klass, mis võtab vastu /translations *endpointi* pihta tehtavaid GET päringuid. Lisaks on olemas *TranslationsService* klass, mille ülesandeks on pärida rakenduse tõlked ja muuta vastus kujule, mida on teegis lihtne kasutada. *TranslationsService* pöördub *LabelApiRepository* klassi poole, mis suhtleb Label API-ga. *LabelApiRepository* teeb GET päringu teenusele /rest/service/label-api/labels (Joonis 6).

```
{
	"labels": [
		{
			"applicationName": "string",
			"key": "string",
			"et": "string",
			"en": "string",
			"ru": "string"
		}
	]
}
```

Joonis 6. GET /labels teenusest tagastatava vastuse formaat JSON kujul.

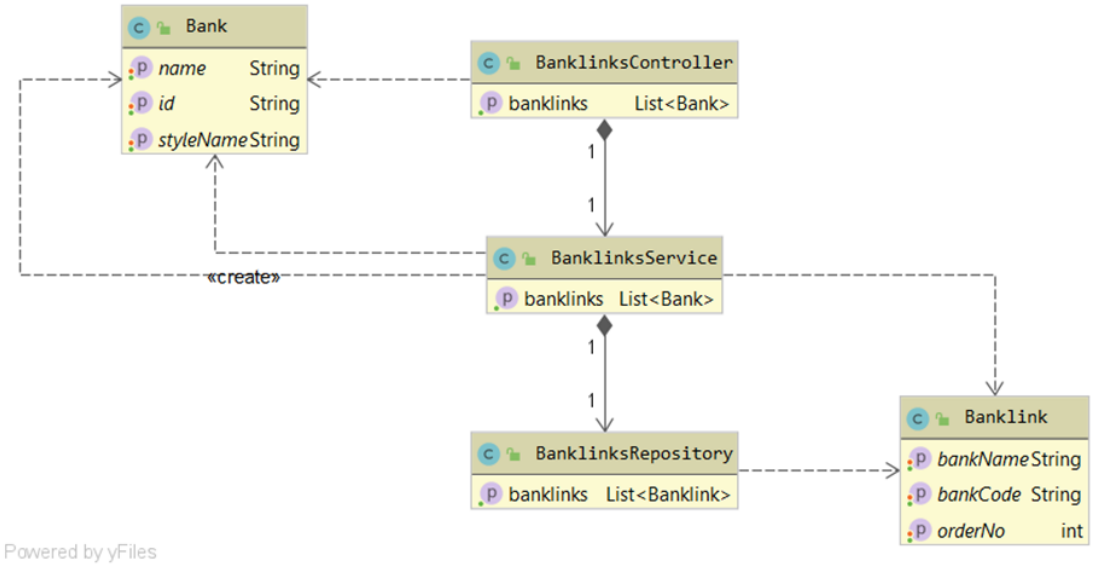
Sellele objekti nimetus rakenduses on *LabelsResource* ja see hoiustab *List<Label>* tüüpi väärtust. *Label* objektis hoitakse rakenduse nime, ühe tõlke võtit ja eesti- ning venekeelseid tõlkeid. *LabelsResource* objektist tehakse *Translations* objekt, milles hoiustatakse ainult eesti- ja venekeelsed tõlkeid võti-väärtus paaridena. GET /labels teenuse tagastab vastusena *Translations* objekti (Joonis 7).

```
{
	"et": {
		"key": "value"
	},
	"ru": {
		"key": "value"
	}
}
```


Joonis 7. GET /translations teenusest tagastatava vastuse formaat JSON kujul.

4.1.2 Pangalinkide nimekirja pärimise teenus

Veebipoolne rakendus peab teadma, milliseid pangalinke välja kuvada. Teenus peab seetõttu tagastama nimekirja kasutatavatest pangalinkidest koos lisaandmetega. Lisaandmetest saab teek olulist infot selle kohta, milline kujundus konkreetsele pangalingi nupule määrata ja millisesse internetipanka pangalink suunab. Teenuse lahendus on kujutatud klassiskeemil (Joonis 8).



Joonis 8. Pangalinkide nimekirja pärimise teenus klassiskeemil kujutatuna.

Rakenduses on *BanklinksController* klass, mis võtab vastu /banklinks teenusele tehtavaid GET päringuid. *BanklinksService* klass sisaldab meetodit, mille ülesandeks on pärida pangalinkide nimekiri ja teisendada need teegi jaoks vastavale kujule. *BanklinksService* klass suhtleb *BanklinksRepository* klassiga, mille kaudu tehakse GET päring Billing API teenusele /rest/Billing/billing-api/banklinks (Joonis 9).

```
[
	{
		"bankCode": "string",
		"bankName": "string",
		"orderNo": "number"
	}
]
```

Joonis 9. GET /rest/Billing/billing-api/banklinks teenusest tagastava vastuse formaat JSON kujul.

Vastus teisendatakse *List<Bank>* objektiks kasutades Java Stream API-t [33], mida saab kasutada Java kollektsioonide puhul. Selle tulemusena kaob vajadus tsüklite järele ja kood on paremini loetav. *List<Bank>* tüüpi vastuse kuju on kujutatud järgmisel joonisel:

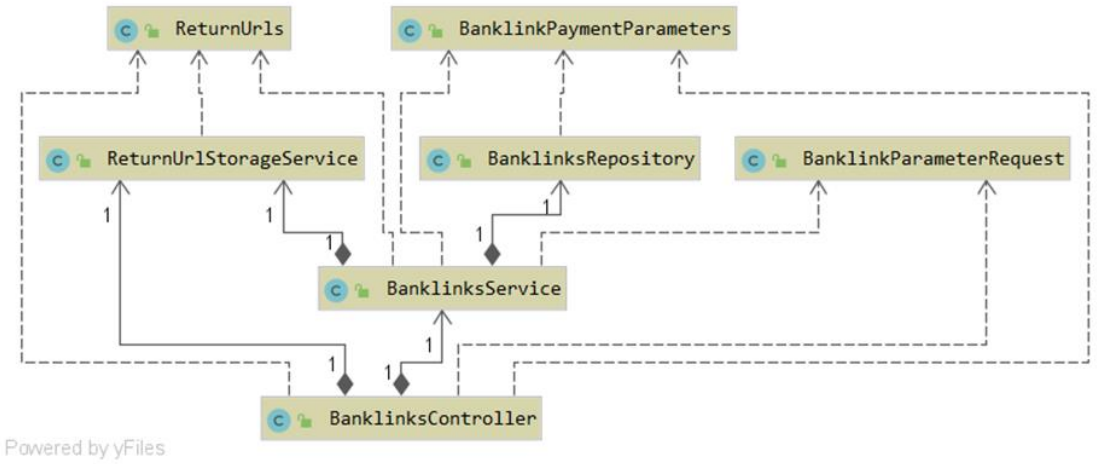
```
[
	{
		"id": "string",
		"name": "string",
		"styleName": "string"
	}
]
```

Joonis 10. GET /banklinks teenusest tagastatava vastuse formaat JSON kujul.

Bank objekti *styleName* väärtus määratakse vastavalt *bankCode* väärtusele. *BanklinksService* klassis on *Map<String, String>* tüüpi muutuja *bankCodeToStyleMapping*, mis hoiustab erinevatele *bankCode* väärtustele vastavaid *styleName* väärtusi.

4.1.3 Maksekorralduse loomise teenus

Rakendus peab oskama suunata klienti internetipanka eeltäidetud maksekorraldusega. Selleks on rakenduses teenus, mis võtab sisendiks makse info ja tagastab vastusena maksekorralduse, mis järgib pangalingi tehnilist spetsifikatsiooni. Teegis loetakse sisse teenuse vastus ning tehakse suunamine internetipanka saadud vastuse põhjal. Teenuse lahendus on kujutatud klassiskeemil (Joonis 11).



Joonis 11. Maksekorralduse loomise teenus klassiskeemil kujutatuna.

BanklinkController klassis on teenus /banklinks/parameters. Teenus võtab vastu POST päringu kehas oleva sisendobjekti. Rakenduses on seda objekti nimetatud kui *BanklinkParameterRequest* (Joonis 12).

```
{
        "bankId": "string",
        "amount": "number",
        "language": "string",
        "referenceNumber": "string",
        "description": "string",
        "successUrl": "string",
        "processingUrl": "string",
        "failedUrl": "string"
}
```

Joonis 12. GET /banklinks/parameters teenuse sisendobjekt JSON kujul.

BanklinksService klassis on meetod, mille ülesandeks on valmistada ette andmed internetipanga maksekorralduse tegemiseks. Kuna internetipangas võib tekkida makse tegemisel erinevaid olukordi, siis on vaja klient vastava olukorra puhul õigele lehe suunata. Nendel lehtedel kuvatakse kliendile maksejärgset teavitust, mille sisu oleneb makse õnnestumisest või ebaõnnestumisest.

ReturnUrlStorageService klass on loodud, et hoiustada suunamise URL-e kasutades Spring Data Redis NoSQL andmebaasimootorit. Rakenduses on *RedisTemplate* funktsionaalsus, mis lihtsustab andmevahetust. Redis on mälusisene andmebaas, milles hoiustatakse andmeid võti-väärtus paaridena. Redise andmebaasi vajadus antud rakenduses põhineb kliendi õigele lehele suunamisel pärast makse sooritamist

internetipangas. Selleks et lehtede URL-ide andmed vahepeal kaduma ei läheksid, siis hoiustamiseks informatsiooni Redise andmebaasis (Joonis 13). Redise andmebaasis olevad andmed asuvad serveri puhvris ja seetõttu on nende andmetega töötamine kiire ja mugav. Klassis on meetodid andmete hoiustamiseks ja lugemiseks. Andmete hoiustamisel luuakse unikaalne võti kasutades Java sisseehitatud UUID abiklassi, mille järgi on võimalik hiljem Redise andmebaasist pärida konkreetse makse juurde kuuluvad suunamise URL-id.

```
"paymentfrontend:data:successurl:token": "successurl"
"paymentfrontend:data:failedurl:token": "failedurl"
"paymentfrontend:data:processingurl:token": "processingurl"
```

Joonis 13. Redise andmebaasis olevad suunamise URL-id võti-väärtus paaridena.

ReturnUrlStorageService klassis luuakse URL väärtus, millele suunatakse klient pärast internetipangas makse tegemist. URL-is määratakse ära ka sõnekirje parameeter *returnToken*, mille väärtus luuakse varasemalt. Suunamise URL *path* koos Redise sessiooni unikaalse identifikaatoriga on järgmisel kujul:

- /payment-frontend/api/banklinks/return?returnToken=unique-redis-token

Edasi luuakse *BanklinkPaymentRequest* objekt, mis põhineb nii päringu kehas olevatele andmetele, üksikutele vaikeväärtustele kui ka äsjaloodud suunamiste URL-idele (Joonis 14).

```
{
	"bankCode": "string",
	"amount": "number",
	"currency": "string",
	"source": "string",
	"refNumber": "string",
	"message": "string",
	"url_return": "string",
	"url_cancel": "string",
	"lang": "string"
}
```

Joonis 14. POST /rest/Billing/billing-api/banklinks/{bankCode}/parameters päringu sisendobjekt JSON kujul.

Parameetrite *url_return* ja *url_cancel* väärtusteks määratakse samaväärne URL. Pank suunab kliendi makse õnnestumise ja ebaõnnestumise korral ühele URL-ile, edasi õigele

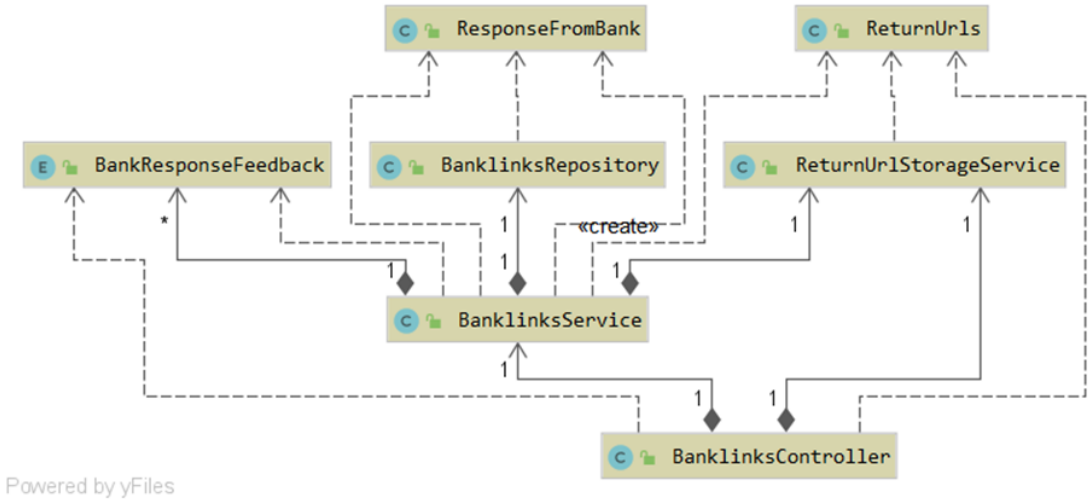
lehele suunamist haldab teek. Seejärel tehakse *BanklinksRepository* klassis POST päring Billing API teenusele /rest/Billing/billing-api/banklinks/{bankCode}/parameters. URL-is oleva *path* parameetri *bankCode* väärtus määratakse *BanklinkPaymentRequest* objektis oleva *bankCode* väärtuse järgi. Päringu kehas edastatakse kogu *BanklinkPaymentRequest* objekti sisu. Teenus tagastab *BanklinkPaymentParameters* objekti (Joonis 15). Vastava objekti edastab teek omakorda internetipangale, millejärgselt toimub kliendi edasisuunamine eeltäidetud maksekorraldusele.


```
{
	"VK_SERVICE": "string",
	"VK_VERSION": "string",
	"VK_SND_ID": "string",
	"VK_STAMP": "string",
	"VK_AMOUNT": "string",
	"VK_CURR": "string",
	"VK_ACC": "string",
	"VK_NAME": "string",
	"VK_REF": "string",
	"VK_MSG": "string",
	"VK_RETURN": "string",
	"VK_CANCEL": "string",
	"VK_DATETIME": "string",
	"VK_MAC": "string",
	"VK_ENCODING": "string",
	"VK_LANG": "string",
	"BANK_URL": "string"
}
```


Joonis 15. POST /banklinks/parameters teenusest tagastatav vastuse formaat JSON kujul.

4.1.4 Internetipangast suunamise teenus

Pärast internetipangas makse sooritamist tuleb klient suunata tagasi õigele lehele. Suunamise URL-id määratakse ära teegis, need omakorda saadetakse /banklinks/parameters teenusele ning neid hoitakse meeles Redise andmebaasis. Makse staatuste järgi on võimalik klienti suunata erinevatele lehtedele, mis võivad näidata infot makse õnnestumise või ebaõnnestumise kohta. Teenuse lahendus on kujutatud klassiskeemil (Joonis 16).



Joonis 16. Internetipangast suunamise teenus klassiskeemil kujutatuna.

BanklinksController klassis on teenus /banklinks/return, mille ülesandeks on töödelda päringusse kaasa antud internetipanga makset ja suunata klient õigele lehele. Esmalt loetakse sisse maksekorralduse parameetrid, mis saadetakse *BanklinksService* meetodile *validateBankResponse*. Antud meetodis luuakse uus objekt *ResponseFromBank*, mille väärtused määratakse päringu parameetrite järgi (Joonis 17).

```
{
        "VK_SERVICE": "string",
        "VK_VERSION": "string",
        "VK_SND_ID": "string",
        "VK_REC_ID": "string",
        "VK_STAMP": "string",
        "VK_T_NO": "string",
        "VK_AMOUNT": "string",
        "VK_CURR": "string",
        "VK_REC_ACC": "string",
        "VK_REC_NAME": "string",
        "VK_SND_ACC": "string",
        "VK_SND_NAME": "string",
        "VK_REF": "string",
        "VK_MSG": "string",
        "VK_T_DATETIME": "string",
        "VK_MAC": "string",
        "VK_ENCODING": "string",
        "VK_LANG": "string",
        "VK_AUTO": "string"
}
```

Joonis 17. POST /rest/Billing/billing-api/banklinks/generateFeedback päringu sisendobjekti formaat JSON kujul.

Järgmisena tehakse POST päring Billing API teenusele /rest/Billing/billing-api/banklinks/generateFeedback, päringu kehas edastatakse *ResponseFromBank* objekti sisu. Lisaks tehakse *BanklinksService* klassi meetodis *validateBankResponse* otsus, millist makse staatust tagastada. *BankResponseFeedback* on *enum* tüüpi klass, milles hoitakse makse staatuseid konstantidena. Kui eelnev päring õnnestus, siis tagastab meetod väärtuse *BankResponseFeedback.SUCCESS*. Samas kui eelneva päringu vastus tagastab erindi, siis määratakse makse staatus vastavalt HTTP päringu staatuskoodile. Staatuskoodide tähendused on klassis määratud *Map<Integer, BankResponseFeedback>* tüüpi andmestruktuuris *feedbackMapping* (Joonis 18):

```
{
        "406": BankResponseFeedback.FAILED,
        "412": BankResponseFeedback.TECHNICAL_ERROR,
        "400": BankResponseFeedback.TECHNICAL_ERROR,
        "500": BankResponseFeedback.TECHNICAL_ERROR
}
```

Joonis 18. FeedbackMapping andmestruktuuri väärtused JSON kujul.

Kui päringu vastuses ilmneb staatuskood, mida pole antud väärtuste hulgas kirjeldatud, siis tagastab meetod erindi *BanklinkException*.

BanklinksController klassis tehakse otsus, et millisele lehele klient suunata. Esmalt laetakse Redise andmebaasist kõik suunamise URL-id kasutades *ReturnUrlStorageService* klassi meetodit *load*. Seejärel tehakse suunamine õigele URL-ile, vastavalt *BanklinksService* meetodi *validateBankResponse* tulemusele.

4.2 Makseliidese teek

Suur osa ettevõttes loodud rakendusi on üles ehitatud React raamistikul. Seetõttu otsustati luua React teek, mida on võimalik teistesse projektidesse lihtsaimal viisil implementeerida.

4.2.1 Rakenduse põhja loomine

Rakenduse põhja loomiseks on käsk *npx create-react-app library*. Käsu tulemusena tekib kaust nimega library ning selle sisse algne projekt, mida on võimalik käivitada ilma lisaseadistusi tegemata. Rakenduse põhi lisati arenduse *feature* harusse. Rakendusele loodi konfiguratsioonifailid. Kuna rakendust ehitatakse Webpacki kaudu, siis tuli selle

tarbeks lisada konfiguratsioonifail *webpack.config.js*. *Package.json* failis on Webpacki sõltuvus, mis võimaldab kasutada Webpacki käske. Kasutades käsureal käsku *npm install* laetakse *node_modules* kausta alla kõik *package.json* failis määratud sõltuvused, et saaks projekti arvutis lokaalselt arendada ja käivitada. Peale sõltuvuste tuli määrata *package.json* failis ära ka näiteks rakenduse nimi, pealkiri, versioon ja kasutatavad käsud. Projekti muudatuste jälgimiseks oma arvutis kasutatakse käsku *npm run watch*, mis kompileerib teegi iga muudatuse peale uuesti.

4.2.2 Rakenduse klassid ja sisendid

Rakenduse põhiosa koosneb kolmest klassist: *BanklinkPayment*, *BanklinkPaymentComponent* ja *BanklinkPaymentTemplate*. Neid klasse võib kujutada ette samamoodi nagu rakenduse äriloogika kihi teenuseid ehk rakendus on loodud kolme eraldi kihina. *BanklinkPaymentTemplate* klassis *renderdatakse* kasutajaliideses olevaid pangalinkide nuppe, summa tekstivälja, rakenduse pealkirja, teavitust kliendile jms. *BanklinkPayment* klassis olev komponent saab algsed sisendid ning saadab need edasi *BanklinkPaymentComponent* klassi. *BanklinkPaymentComponent* on vahenduslüli kahe klassi vahel, milles valmistatakse andmed ette kasutajaliidese tarbeks.

Teegis on kaust *components*, milles asub *BanklinkPayment* komponent. Klassis olevat komponenti saavad kasutada kõik React projektid, mis *importivad* antud teeki. Komponent saab kohustusliku *propsi* ehk sisendina *referenceNumber* ehk viitenumbri väärtuse. Samuti on kohustuslikud määrata *successUrl*, *failedUrl* ja *processingUrl* väärtused, mida kasutatakse kliendi suunamiseks õigele URL-ile pärast pangas makse tegemist. Lisaks on sisenditena võimalik määrata väärtus *environment* ehk valik kolme erineva keskkonna vahel, *headingLabel* ehk rakenduses kuvatav pealkiri, *initialAmount* ehk kliendi poolt makstav summa, *language* ehk rakenduse keelsus ja *amountFieldLabel* ehk summa tekstiväljas kuvatav tekst. Kui kliendile tahetakse kuvada teavitust, siis määratakse ära ka *noticeType* ja *noticeLabel* väärtused, mis määravad ära teavituse stiili ja kuvatava sõnumi.

4.2.3 Abistavate klasside loomine

ApiService klassisiseselt toimuvad päringud projekti *backendi* teenustele Axios HTTP kliendi kaudu. *ApiService* klassis päritakse pangalinke, rakenduse tõlkeid, valmistatakse andmed ette maksekorralduse loomiseks ning suunatakse klient panka maksma.

Komponendi *props* väärtused antakse sisendiks *BanklinkPaymentComponent* klassis olevale komponendile. Selle esmasel loomisel päritakse *ApiService* klassist rakenduse tõlked ning määratakse ära summa tekstiväljas olev väärtus. *BanklinkPaymentComponent* saadab *BanklinkStore* klassi kaudu päritud pangalinkide listi *BanklinkPaymentTemplate* komponendile edasi. *BanklinkStore* klassis hoitakse pangalinkide objektide listi ning lisaks on meetod kliendi panka suunamiseks. *Store* tüüpi klassides hoiustatakse rakenduse ajas muutuvaid andmeid. Reeglina on nendes klassides ka klassisiseste andmete muutmisega seotud meetodid. *RootStore* klass luuakse *BanklinkPayment* klassi konstruktoris, mis on meetod, mis käivitatakse komponendi loomisel. *RootStore* klassis initsialiseeritakse kõik *store* tüüpi klassid. Teegi komponendis *BanklinkPayment* kasutatakse abiklassi *Provider*, mis võimaldab kõikidel sisemistel komponentidel pääseda ligi *store* tüüpi klassidele, mis on loodud *RootStore* klassis. Komponendid pääsevad ligi projektis olevatele *store* klassidele ning nende sees olevatele muutujatele ja meetoditele lisades klassi algusesse MobX-i annotatsiooni *@inject*.

4.2.4 Teegi üleslaadimine NPM salve

Teegis on konfiguratsioonifailid paketi ehitamiseks ning üleslaadimiseks. Skriptifailid käivitatakse pärast Bamboo keskkonnas rakenduse edukat ehitamist.

Build.sh skriptifailis laetakse sõltuvused alla käsuga *npm ci*, mis on sarnane käsule *npm install*, kuid sõltuvused laetakse alla *package-lock.json* faili põhjal [34]. Seejärel ehitatakse pakk käsuga *npm build*, mis kompileerib rakenduse failid, et neid oleks võimalik teistes projektides kasutada. Lõpuks käivitatakse käsk *npm run test*, mis jooksutab üksuste testid, mis peavad täies mahus läbima. Ebaõnnestunud testide korral pakki üles ei laeta.

Publish.sh skriptifaili jooksutamisel laetakse pakk üles ettevõtte privaatsesse Artifactory NPM pakkide salve. Skriptifail nõuab kahte sisendit – *branch* ja *auth_token*. *Branch* ehk haru väärtusena on lubatud kasutada ainult *master* haru, kuna sinna laetakse üles ainult valmis ja koodiülevaatuse läbinud koodi. *Auth_token* on väärtus, mille annab ette Artifactory keskkond. Seda kasutatakse paki üleslaadimisel kasutaja autentimiseks. Skripti käivitamisel tekib projekti *.npmrc* fail, kus on määratud *registry* muutuja, mille väärtus peab olema privaatse Artifactory NPM registri URL koos *auth_token* väärtusega. NPM loeb *.npmrc* failis oleva muutuja sisse ning määrab registri, kuhu pakki üles laadida. Makseliidese teek laetakse üles NPM registris olevasse alamkataloogi *npm-local*.

4.3 Makseliidese mikroveeb

Mikroveeb on loodud mitte-React rakenduste tarbeks. Veebis kuvatakse pangalinkide makseliidest. Mikroveeb kasutab makseteenuste teeki, mis *imporditakse package.json* failis privaatsest salve NPM-i kaudu. Komponent saab väärtused kasutades URL-i sõnekirje parameetreid. Parameetrite väärtused loetakse sisse *window* objekti *location.search* väärtuse küljest. Lisaks kasutatakse teeki *qs* [35], mis loeb sisse parameetrite väärtused kasutades *parse* meetodit, millega on võimalik parameetrite väärtusi lihtasti lugeda.

Development, test ja *live* keskkondade jaoks on loodud eraldi URL-idel paiknevad mikroveebid. Bamboo serveris on kolm eraldi keskkonda, kuhu on võimalik mikroveebi paigaldada. Paigaldamise käigus tõmmatakse all artifaktid ehk programmikoodi osad, mis luuakse automaatselt pärast koodi harusse üleslaadimist ja testide läbimist. Bamboos laetakse automaatselt alla *frontend-swift* ja *frontend-build* artifaktid, seejärel jooksutatakse *upload.sh* skriptifail mikroveebi üleslaadimiseks Swifti kaudu. Skriptifailile antakse ette keskkonna ja Swift *authorization token* argumentide väärtused. Skriptifailis luuakse avalik konteiner, kui seda pole varem loodud. Lisaks laetakse konteinerisse üles *frontend-build* artifaktis olevad failid. Pärast skripti jooksutamist on mikroveeb paigaldatud. *Frontent-swift* artifaktist loetakse skriptifailid ja konfiguratsioonid, mis on loodud Swifti ülesannete täitmiseks.

4.4 Liidese implementatsiooni näited

Järgnevalt on toodud makseliidse kasutamise näited nii React kui ka mitte-React veebides.

4.4.1 React teegi näide

React raamistikul põhinevates rakendustes on liidest võimalik NPM-i kaudu projekti installeerida käsuga *npm install*. Käsk lisab *package.json* faili uue paketisõltuvuse (inglise keeles *dependency*), mille korral laetakse vastav pakett alla ning lisatakse *node_modules* kausta. Alternatiivina on võimalik *package.json* faili sõltuvuste nimekirja lisada rida *"payment-react-component": "^1.0.6“*. 1.0.6 tähistab versiooninumbrit (*semantic version*), mida tahetakse kasutada ning ^ märki versiooninumbri ees kasutatakse uuema paki installeerimiseks selle olemasolul. Seejärel saab teeki React

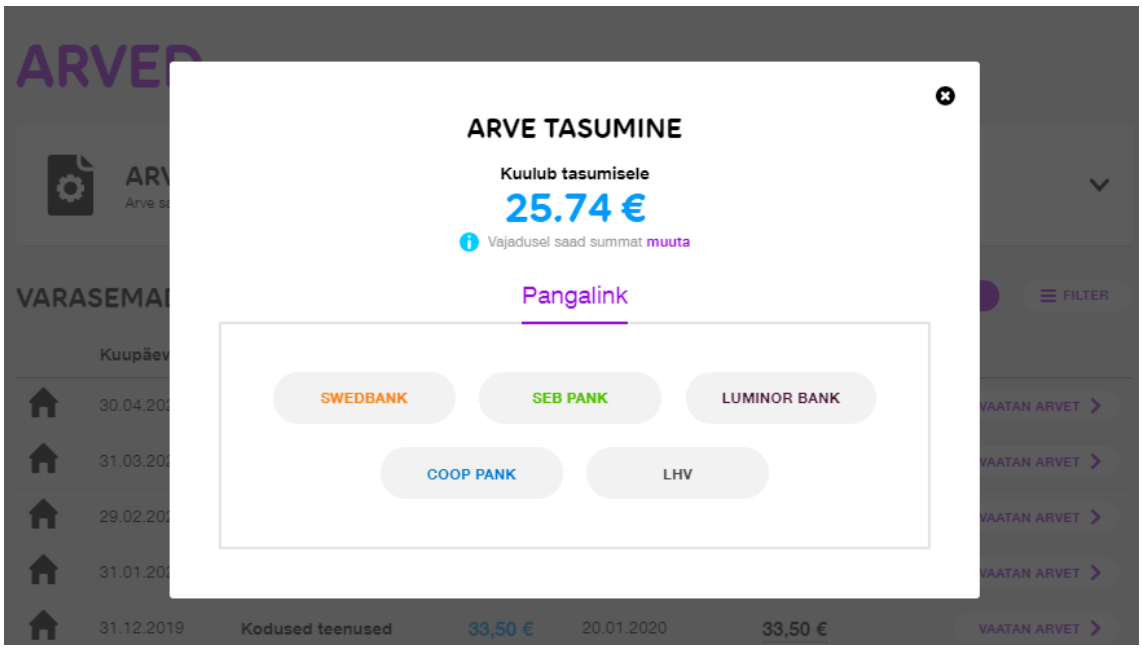
projektides kasutusele võtta kasutades teegi *importimist*. Rakendusse tuleb lisada *BanklinkPayment* komponent koos sinna kuuluvate parameetritega (Joonis 19).

```
import { BanklinkPayment } from 'payment-react-component';

public render() {
	return (
		<BanklinkPayment
			initialAmount={25}
			referenceNumber="1234567890"
			successUrl="https://www.*.ee/payment-successful"
			processingUrl="https://www.*.ee/payment-processing"
			failedUrl="https://www.*.ee/payment-failed"
			amountFieldLabel="Insert amount here"
			headingLabel="Payment"
			language="et"
			environment="dev"
			noticeLabel="Notice text"
			noticeType="warning"
			onRedirectToBank={banklinkName =>
			someFunction(banklinkName)}
		/>
	);
}
```

Joonis 19. Teegi kasutamise programmikoodi näide.

Sellisel meetodil põhineva lahenduse implementatsioon on ettevõtte iseteeninduse veebis (Joonis 20). Makseliidest kasutatakse arvete tasumise eesmärgil.



Joonis 20. Pangalinkide makseliides ettevõtte iseteeninduses.

4.4.2 Mitte-React komponendi näide

Mitte-React veebid saavad makseliidese võtta kasutusele URL-i kaudu. URL-id on keskkonnapõhised ja URL-is on kohustuslik määrata päringusõne parameetrite väärtused.

Kohustuslike parameetrite hulka kuuluvad:

- referenceNumber – arve viitenumber
- successUrl – URL, kuhu suunatakse pärast edukat makse sooritamist
- processingUrl – URL, kuhu suunatakse, kui makse on töötlemisel
- failedUrl – URL, kuhu suunatakse pärast katkestatud või ebaõnnestunud makset

Valikulised parameetrid on järgmised:

- initialAmount (vaikeväärtus 0,00)
- language (vaikeväärtus 'et')
- amountFieldLabel (vaikeväärtus 'Makse summa')
- headingLabel (vaikeväärtus 'Teen makse')
- noticeLabel (vaikeväärtus puudub)
- noticeType (vaikeväärtus 'warning')

Koostatakse URL, milles peavad olema väärtustatud kõik kohustuslikud parameetrid. Kohustuslikel parameetritel puuduvad vaikeväärtused. Lisaks võib anda väärtused ka valikulistele parameetritele. URL-i võib avada brauseris tavalise veebilehena või näiteks kasutada HTML iframe komponendina lehe seesmiselt (Joonis 21).

```
<iframe src="https://iseteenindus-dev.*.ee/payment
    ?initialAmount=25
    &referenceNumber=1234567890
    &language=et
    &successUrl=https://www.*.ee/payment-successful
    &processingUrl=https://www.*.ee/payment-processing
    &failedUrl=https://www.*.ee/payment-failed
    &headingLabel=Ettemakse
    &noticeType=warning
    &noticeLabel=Teavituse%20sisu
    &amountFieldLabel=Sisesta%20ettemaksu%20summa%20siia">
</iframe>
```

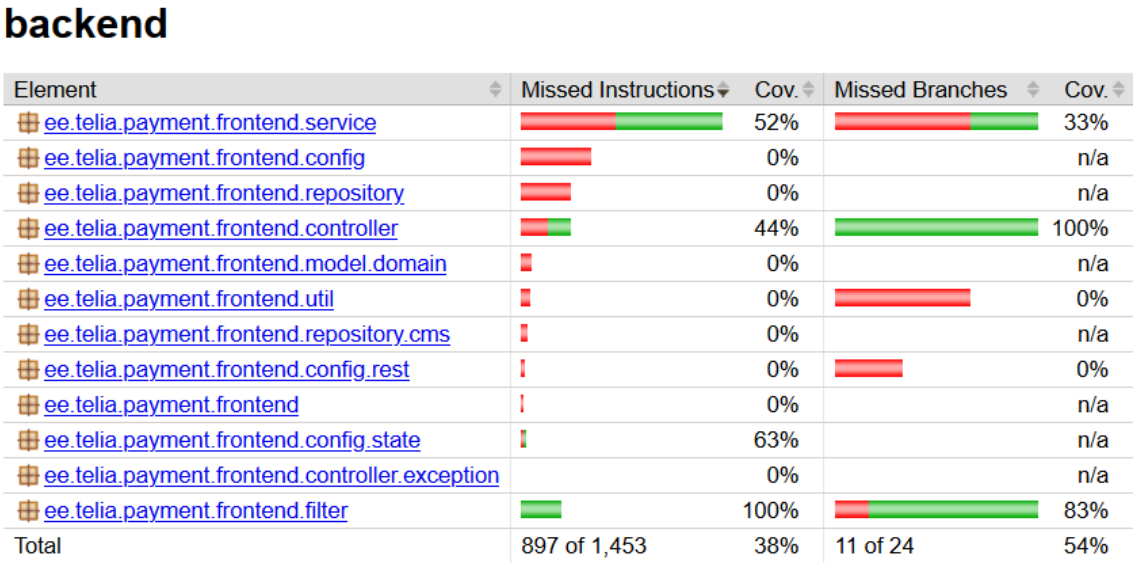
Joonis 21. Mitte-React veebilehtedes makseliidese implementeerimise koodinäide DEV keskkonna URL-i näitel kasutades iframe komponenti.

4.5 Lahenduse kontroll ja testimine

Testid on mõeldud projektis vigade avastamiseks. Testid tagavad projektis koodi terviklikkuse. Üksuste testid kontrollivad komponentide funktsionaalsust. Samuti on töösse kaasatud individuaalseid testijaid, kes otsisid vigu ning kontrollisid, et rakendus vastaks ettenähtud nõuetele. *Feature* harudele tehakse *pull requeste*, et teised arendajad saaksid loodud programmikoodist leida vigu ning valideerida koodi korrektsust. JIRA keskkonnas määratakse käsiloleva *story* staatuseks *systemtest done* pärast üksuste testide tegemist ning oma arenduse kontrollimist. *Story* saab staatuseks *acceptance test done*, kui ettevõttesisene testija on lahenduse üle kontrollinud. Pärast koodi *live* keskkonda paigaldamist on võimalik *story* lõpetada ehk muuta staatuseks *closed*.

React raamistikul loodud mikroveebis ning teegis on kasutuses üksuste testide loomiseks Jest [36] testraamistik. Raamistiku seadistuste tegemiseks on *package.json* failides Jesti konfiguratsioonid, kus on määratud testifailide failitüübid ning testimisse kaasatud kaustad. Mikroveebis on kuus üksuse testi, mille hulgas testitakse, kas parameetrite väärtused jõuavad korrektselt URL-i päringusõnedest makseliidese komponendi sisenditesse. Testidega oli kogu funktsionaalsus kaetud. Testide katvuse nägemiseks käivitatakse käsureal käsk *npx jest --coverage*, mis loob uue kausta nimega *coverage*, mille sees olev *index.html* fail võimaldab vaadata testide katvust nii kokkuvõtvalt kui ka iga klassi põhiselt. Teegi testimiseks on üksuste testid, mis testivad nii makseliidese komponente, *store* klasse kui ka abistavate klasside meetodeid. Testide katvus jäi 70% piirile. Äriloogika kihis on kasutuses üksuste testide tegemise testraamistik JUnit [37]. Testide koodikatvuse nägemiseks kasutatakse Jacoco [38] teeki, mis sarnaselt Jest

testraamistikule loob uue kausta koos *index.html* failiga, mis tuleb veebibrauseris avada. *Backend* teenuste testide koodikatvus oli 38% (Joonis 22).



backend

Element	Missed Instructions	Cov.	Missed Branches	Cov.
ee.telia.payment.frontend.service		52%		33%
ee.telia.payment.frontend.config		0%		n/a
ee.telia.payment.frontend.repository		0%		n/a
ee.telia.payment.frontend.controller		44%		100%
ee.telia.payment.frontend.model.domain		0%		n/a
ee.telia.payment.frontend.util		0%		0%
ee.telia.payment.frontend.repository.cms		0%		n/a
ee.telia.payment.frontend.config.rest		0%		0%
ee.telia.payment.frontend		0%		n/a
ee.telia.payment.frontend.config.state		63%		n/a
ee.telia.payment.frontend.controller.exception		0%		n/a
ee.telia.payment.frontend.filter		100%		83%
Total	897 of 1,453	38%	11 of 24	54%

Joonis 22. Backend teenuste üksuste testide katvus.

5 Järeldused ja projekti edasised arengud

Programmeeritud lahendus täitis oma eesmärki. Töö käigus valmis toimiv pangalingimaksete lahendus, mida on võimalik kasutada teistes projektides eraldiseisva komponendina. Pangalingi maksed on testitud programmeerija poolt loodud üksuste testide ja individuaalsete testijate kaudu. Testide katvust rakenduse *backend* rakenduses saaks suurendada luues rohkem teste. Pangalinkide makseliidest on võimalik kasutada nii React kui ka mitte-React veebides. Töö käigus valminud makseliides võeti kasutusele ettevõtte iseteeninduses arvete maksmise eesmärgil. Lisaks kasutatakse projekti ühes mitte-React veebis, mis kasutab makseliidest mikroveebi kaudu.

Üheks projekti edasiseks eesmärgiks on komponendi brändipõhise lahenduse loomine, mille käigus on võimalik muuta komponentide kujundust vastavalt veebikeskkonna brändile. Lisaks on eesmärgiks liidestada alternatiivseid makseviise, mis võimaldaksid klientidel peale pangalinkide maksta ka kasutades mTasku rakendust või krediitkaarti. Lõpuks on planeeritud töödelda SSO (*Single Sign-On*) küpsistes olevaid kasutajaandmeid. Igale sisselogitud kliendile luuakse brauseris SSO küpsis, mis hoiustab JWT (*JSON Web Token*) sessiooni *token* väärtust. JWT järgi on võimalik klienti hiljem tuvastada, kui tuleb lahendada kliendipöördumisi. JWT järgi on võimalik süsteemi logidest leida makse päringud ja seostada neid konkreetse kliendiga.

6 Kokkuvõte

Töö eesmärgiks oli luua makseteenuste liides, mis võimaldab makseid teha pangalinkide kaudu. Pangalink võimaldab makset teha eeltäidetud maksekorraldusega, mis valmistatakse ette enne rakendusest panka edasi suunamist. Kliendile ei võimaldata pangas enam andmete muudatusi teha. Uus lahendus vähendab oluliselt muudatuste tegemisele ja vigade parandamisele kuluvat ressurssi. Makseteenuste liides muutub tsentraalseks süsteemiks, mida on võimalik teistesse projektisesse implementeerida. Lahendus on kasutatav nii React raamistikul ehitatud veebides kui ka mitte-React veebides. Liides peab järgima analüüsi käigus loodud arhitektuuri.

Esmalt uuriti varasemat pangalinkide makseteenuse lahendust, mis oli ettevõttes ühes projektis kasutusel. Selle põhjal hakati looma uut lahendust. Bitbucket keskkonnas loodi uus koodihoidla nimega payment-frontend. Teenused loodi põhinedes Spring Boot raamistikul üles ehitatud äriloogika kihile. Äriloogika kiht koosneb neljast teenusest. Teenused pöörduvad väliste API-de poole. Billing API kaudu küsitakse pangalinkide andmed, valmistatakse ette maksekorralduse andmed ning tehakse suunamine õigele URL-ile pärast pangast saadud makse kinnitust. Label API kaudu päritakse rakenduse tarbeks nii eesti- kui ka venekeelsed tõlked.

Järgmiseks loodi React raamistiku põhjal pangalinkide makseliidese teek, mille korral kuvatakse kliendile pangalinke ning makse summat. Programmeerimiskeelena on kasutatud Microsoft arendatud TypeScript programmeerimiskeelt, mis võimaldas muutujate tüüpe annoteerida. Teek kasutas ettevõtte privaatses salves olevat ettevõtte UI/UX (*User Interface/User Experience*) tiimi loodud React UI teeki, mida kasutati kasutajaliidese mallis nuppude, tekstivälja ja muude komponentide loomiseks. Teek pöördub projektis loodud äriloogika kihi teenuste poole. Muutujaid hoiustatakse *store* tüüpi klassides kasutades MobX teegi funktsionaalsust. Teek laeti üles privaatsesse Artifactory NPM salve, mis võimaldab teistel projektidel antud teeki kasutusele võtta.

Lisaks loodi eraldiseisev mikroveeb rakenduste tarbeks, mis ei ole üles ehitatud React raamistiku põhjal. Mikroveeb on ehitatud React raamistikul ning implementeerib

pangalinkide makseteenuse teeki. Teek loeb muutujad sisse URL-i sisse kirjutatud sõnekirje parameetritest, mida kasutatakse makseteenuse komponendi sisenditena.

Nii äriloogika kihi, presentatsioonikihi kui ka teegi puhul kirjutati automaattestid, mis pidid edukalt läbima paki ehitamise käigul. Vigaste testidega pakki ei ole lubatud paigaldada. Lisaks kontrollisid mitmed ettevõttes töötavad testijad komponentide vastavust ettenähtud nõuetele. Enne koodi üles laadimist tehti koodiülevaatust, kus kogenenumad arendajad andsid tagasisidet ning mille käigus sai jooksvalt vigu parandatud.

Valminud töö vastas määratud eesmärkidele ning loodud liidest on võimalik kasutada teistes rakendustes välise sõltuvusena. Projekti edasise arenduse jaoks on planeeritud uute makseviiside liidestamine ja komponentide brändipõhiseks muutmine.

Kasutatud kirjandus

[1] "Docker," [WWW]. Available: https://et.wikipedia.org/wiki/Docker. (20.05.2020).

[2] "Vikipeedia, "HTTP-küpsis"," [WWW]. Available: https://et.wikipedia.org/wiki/HTTP-k%C3%BCpsis. (24.05.2020).

[3] "Vikipeedia, "Proksiserver"," [WWW]. Available: https://et.wikipedia.org/wiki/Proksiserver. (23.05.2020).

[4] "Vikipeedia, "Toru (arvutustehnika)"," [WWW]. Available: https://et.wikipedia.org/wiki/Toru_(arvutustehnika). (23.05.2020).

[5] "IntelliJ IDEA," [WWW]. Available: https://www.jetbrains.com/idea/. (22.05.2020).

[6] "Git," [WWW]. Available: https://git-scm.com/. (22.05.2020).

[7] "React," [WWW]. Available: https://reactjs.org/. (22.05.2020).

[8] "Spring Boot," [WWW]. Available: https://spring.io/projects/spring-boot. (22.05.2020).

[9] "Atlassian Bitbucket," [WWW]. Available: https://bitbucket.org/. (22.05.2020).

[10] "NPM," [WWW]. Available: https://www.npmjs.com/. (22.05.2020).

[11] "Artifactory," [WWW]. Available: https://jfrog.com/artifactory/. (22.05.2020).

[12] "Kubernetes," [WWW]. Available: https://kubernetes.io/. (21.05.2020).

[13] "OpenStack Swift," [WWW]. Available: https://wiki.openstack.org/wiki/Swift. (22.05.2020).

[14] "Differences between active-active and active-passive cluster," [WWW]. Available: https://www.sebastien-han.fr/blog/2012/05/26/differences-between-active-active-and-active-passive-cluster/. (23.05.2020).

[15] "Domeeninimede süsteem," [WWW]. Available: https://et.wikipedia.org/wiki/Domeeninimede_s%C3%BCsteem. (21.05.2020).

[16] "Nginx," [WWW]. Available: https://www.nginx.com/. (22.05.2020).

[17] "Atlassian JIRA," [WWW]. Available: https://www.atlassian.com/software/jira. (22.05.2020).

[18] "Eesti Pangaliit," [WWW]. Available: https://www.pangaliit.ee/et. (22.05.2020).

[19] "Pangalingi spetsifikatsioon," [WWW]. Available: https://pangaliit.ee/arveldused/pangalingi-spetsifikatsioon. (22.05.2020).

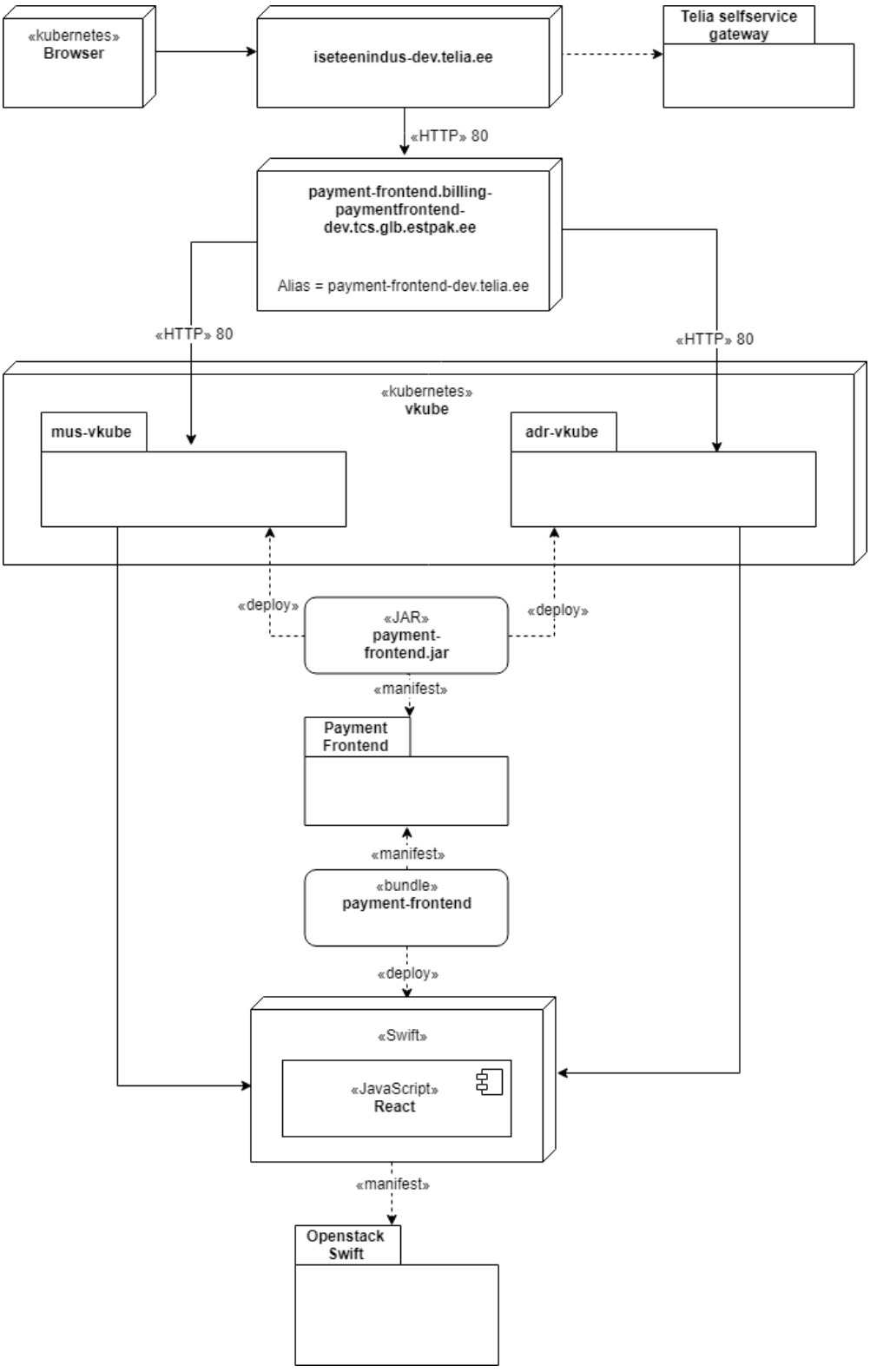
[20] "Atlassian Bamboo," [WWW]. Available: https://www.atlassian.com/software/bamboo. (23.05.2020).

[21] "Continuous integration vs. continuous delivery vs. continuous deployment," [WWW]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. (22.05.2020).

[22] "Kubernetes concepts," [WWW]. Available: https://kubernetes.io/docs/concepts/. (23.05.2020).

[23] "Containers at Google," [WWW]. Available: https://cloud.google.com/containers/. (22.05.2020).
[24] "Gradle Build Tool," [WWW]. Available: https://gradle.org/. (22.05.2020).
[25] "TypeScript," [WWW]. Available: https://www.typescriptlang.org/. (22.05.2020).
[26] "Webpack," [WWW]. Available: https://webpack.js.org/. (22.05.2020).
[27] "MobX," [WWW]. Available: https://mobx.js.org/. (22.05.2020).
[28] "Axios," [WWW]. Available: https://www.npmjs.com/package/axios. (22.05.2020).
[29] "Spring," [WWW]. Available: https://spring.io/. (22.05.2020).
[30] "Redis," [WWW]. Available: https://redis.io/. (22.05.2020).
[31] "Spring Initializr," [WWW]. Available: https://start.spring.io/. (22.05.2020).
[32] "X-tee mustrikataloog," [WWW]. Available: https://moodle.ria.ee/mod/page/view.php?id=245. (23.05.2020).
[33] "Java Stream API," [WWW]. Available: https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html. (22.05.2020).
[34] "npm-ci," [WWW]. Available: https://docs.npmjs.com/cli/ci.html. (22.05.2020).
[35] "qs," [WWW]. Available: https://www.npmjs.com/package/qs. (23.05.2020).
[36] "Jest," [WWW]. Available: https://jestjs.io/. (22.05.2020).
[37] "JUnit4," [WWW]. Available: https://junit.org/junit4/. (22.05.2020).
[38] "Jacoco," [WWW]. Available: https://www.eclemma.org/jacoco/. (22.05.2020).

Lisa 1 – Rakenduse paigalduse arhitektuurivaade



Lisa 2 – Kubernetese klastrite vaade