TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Sven Kadak 134293IAPB

EMBEDDABL: VEEBILEHTE SÜSTITAV KOODI JOOKSUTAMISE PLATVORM

Bakalaureusetöö

Juhendaja: Martin Rebane
MSc

Tallinn 2017

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: Sven Kadak

22.05.2017

Annotatsioon

Käesoleva töö eesmärk on luua platvorm programmide lähtekoodi jooksutamiseks. Töös realiseeritakse kaks komponenti. Esiteks luuakse intuitiivne veebilehte süstitav pistikrakendus, mis laseb koodi lihtsasti jagada ja käivitada. Koodi jooksutava kasutaja jaoks on programmi käivitamine täielikult abstraheeritud ning edasi-tagasi kopeerimise asemel saab koodi jooksutada ühe klikiga veebilehelt lahkumata. Teise komponendina luuakse serverilahendus, mis vastutab programmide jooksutamisega seonduvate tegevuste eest.

Käesoleva töö teevad võimalikuks mitmed tehnoloogiad, mis on viimase paari aasta jooksul jõudsalt arenenud. Rakendus luuakse Vue.js *front-end* raamistikku kasutades. Serveri põhilisteks alustehnoloogiateks on turvalised ja kerged LXC konteinerid, OverlayFS ja BTRFS failisüsteemid, Node.js, Express veebiraamistik, Websocketid ning Ubuntu operatsioonisüsteem.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 29 leheküljel, 6 peatükki, 15 joonist.

Abstract

Embeddabl: A Platform That Enables Code Execution on Websites

The purpose of this thesis is to showcase a platform that enables users to execute the source code of programs on websites. The platform consists of two components both of which are implemented: an application that can be embedded into websites and a server-side solution that deals with everything related to executing the programs.

The idea is motivated from the fact that to this day people still use static text blocks to share code online. Copying code between browser and the host system is firstly inconvenient and secondly the code may depend on many other programs that the user might not have installed or lacks skills to install them.

The implemented platform abstracts away executing the embedded code: instead of copying text back and forth it requires only a single click to do so without having to leave the web page.

This study is made possible by several technologies that have improved thrivingly over the last couple of years. The embeddable application is created using Vue.js front-end framework. Main technologies used to create the server-side solution are secure and lightweight LXC containers, OverlayFS and BTRFS filesystems, Node.js, Express web framework and Ubuntu operating system.

The thesis is in Estonian and contains 29 pages of text, 6 chapters, 15 figures.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface* (rakendusliides)
BTRFS	Failisüsteem, mis toetab *Copy-on-Write* tehnoloogiat, tehes sellest üliefektiivse failisüsteemi, ning võimaldab kasutajate kõvaketta kasutusele panna dünaamilise piirangu. [11]
CDN	*Content Delivery Network* (sisuedastusvõrk)
Copy-on-Write	Efektiivne ressursside haldamise tehnika, mille põhjal ei ole vaja luua uut ressurssi duplikeeritud ressursist, mida pole muudetud [1] . Otsetõlkes: „kopeeri, kui on muudetud”.
CSS	*Cascading Style Sheets* (kaskaadlaadistik)
DoS-rünne	*Denial of Service* (teenusetõkestamise) rünne
GIF	*Graphics Interchange Format*. 1987. aastal loodud veebis laialt kasutatav rasterpildi vorming, mis toetab animatsioone. [2]
HTML	*HyperText Markup Language* (hüperteksti märgistuskeel)
HTTP	*Hypertext Transfer Protocol* (hüperteksti edastusprotokoll)
I/O	*Input/output* (sisend/väljund)
JOOP	TTÜ kursus „Objektorienteeritud programmeerimine Javas”
JSON	*JavaScript Object Notation* (Javascripti objekti noteering)
RAM	*Random-access memory* (muutmälu)
SaaS	*Software as a service* (tarkvara teenusena)
TCP	*Transmission Control Protocol* (edastusohje protokoll)
UX	*User Experience* (kasutajakogemus)
YAML	*YAML Ain't Markup Language* (YAML pole märgistuskeel)

Sisukord

Jooniste loetelu

1 Sissejuhatus

Üheksakümnendate lõpus, mil veebi leiutamisest oli möödunud nii mõnigi aasta, olid veebilehed jätkuvalt üdini staatilised: puudusid animatsioonid (kui GIF-id välja arvata) ja ilusad visuaalid. Web 2.0 kontseptsiooniga on eelnimetatud lihtsusest saanud ajalugu ning uueks suunaks on võetud modernse veebielamuse loomine interaktiivsusele ning animeeritusele rõhudes. Selle ülemineku varjus on tavapärased veebielemendid asendunud eraldiseisvate teenustega, mille võtab kokku nö *plug–and–play* lähenemine ehk lahendused töötavad kohe "karbist välja võttes" ning eraldi seadistamist vajamata. Nii on kommentaariumite jaoks nüüd Disqus, videoid kuvatakse otse YouTube'ist jne.

1.1 Eesmärgid

Käesolev bakalaureusetöö võtab eesmärgiks veebi moderniseerimisse panustada ühe vananenud elemendi, staatiliste programmi lähtekoodi tekstikastikeste, väljavahetamisega. Valmiva *SaaS* lahenduse peamiseks sihtgrupiks on koodi jagada tahtvad inimesed (programmeerimise õpetuste tegijad, õppejõud jm), kes saaksid õpetuse, artikli või lihtsalt lähtekoodi põhimõtet otse brauseris, lehelt lahkumata, interaktiivselt illustreerida.

Sellise lahenduse boonused staatiliste tekstikastidega võrreldes on kasutajapoolne võimalus koodi muuta ning kindlus, et kood töötab. Praegu võib ennast tihti leida olukorrast, kus koodi oma arvutis töölesaamine sõltub teiste programmide paigaldamisest, mis esiteks ei pruugi töötada ning teiseks võib osutuda ajakulukaks.

Töö sisulisteks eesmärkideks on

- intuitiivse veebilehte süstitava pistikrakenduse loomine, mis võimaldab kasutajatel hõlpsasti lisatud koodijupikestega tutvuda ja neid reaalajas jooksutada;

- serverilahenduse väljatöötamine, mis suudab kasutajate määratud koodi jooksutada;
- HTTP liidese loomine, mida saab kasutada Tallinna Tehnikaülikooli kursuse „Objektorienteeritud programmeerimine Javas” (JOOP) jaoks loodavas rakenduses tudengite koodi käivitamiseks (see rakendus ei ole käesoleva töö osa).

1.2 Võrdlus sarnaste lahendustega

Oma olemuselt eristub loodav rakendus teistest sarnastest lahendustest selle poolest, et keskendutakse ühe asja tegemise peale, milleks on koodi jooksutamise võimaldamine. Loodav lahendus üritab samaaegselt rahuldada nii koodihalduri erivajadusi kui ka nõudlike lõppkasutajate soove. Teised lahendused on omaette eksisteerivad platvormid, mille eesmärgiks ei ole keskkonna veebilehte süstimine. Samuti piirduvad olemasolevate lahenduste nagu CodePicnic[1] poolt pakutavad võimalused *iframe* HTML elemendi süstimisega lehte, mis seab mõlema kasutaja – koodihalduri ja jooksutaja – kogemusele selged piirangud.

Käesoleva töö saadus ei suru peale ühte kindlat võimalust koodi lisamiseks, vaid laseb kasutajal valida meelepärasema: kas otse oma serverist või loodava platvormi serverist; kas teksti (JSON) kujul, giti repositooriumist, ZIP arhiivist või üldsegi HTTP liidest kasutades. Olemasolevad lahendused taolisi automatiseeritavaid tegevusi ei realiseeri.

1.3 Töö struktuur

Töö koosneb neljast osast. Esmalt defineeritakse rakenduse nõuded, millele järgneb nõuetel baseeruva rakenduse loomise protsessi ja kasutamise kirjeldus. Seejärel kirjeldatakse serverilahendusena valmivate Websocketi ja HTTP veebiliideste loomist ning koodi jooksutamist võimaldavate komponentide rakendamist. Lõpuks analüüsitakse loodud süsteemi koormusele vastupidamist.

[1] https://codepicnic.com/

2 Platvormi spetsifikatsioon

Käesolev peatükk kirjeldab loodava kliendipoolse rakenduse mittefunktsionaalseid ja funktsionaalseid nõudeid ning kõrgtaseme arhitektuuri, millel loodav platvorm põhineb.

2.1 Pistikrakenduse mittefunktsionaalsed nõuded

Mittefunktsionaalsete nõuete alla käivad näiteks kasutatavust, toetatavust puudutavad küsimused [3] .

Kasutatavuse seisukohast on suurim roll loodaval pistikrakendusel ning HTTP API-l. Mõlemad peavad olema loodud olemasolevate lahenduste põhimõtteid järgides, sest nendega on kasutajad harjunud. Innovaatilise kasutajakogemuse loomine ei ole selle töö skoop. Näiteks on rakenduse puhul oluline erinevate komponentide loogiline paigutus kasutajaliideses.

Toetatavust silmas pidades on oluline hinnata, kes on loodava lahenduse lõppkasutajad. Kuna käesoleval juhul on nendeks tehniliselt pädevamad inimesed, ei ole mõtet võtta eraldi eesmärgiks toetada vanemaid brausereid kui Internet Explorer 11.

2.2 Pistikrakenduse funktsionaalsed nõuded

Funktsionaalsed nõuded kirjeldavad, millist funktsionaalsust rakendus pakub ning kuidas rakendus vastavalt kasutajapoolsele sisendile käitub [3] . Funktsionaalsed nõuded täidavad tähtsat rolli programmi arendamise etapis: tehtud peab saama spetsifikatsioonile vastav tarkvara. Järgnevalt on toodud nimekiri realiseeritava kliendipoolse rakenduse funktsionaalsetest nõuetest:

1. Kui lisatud on vaid üks fail, kuvada lihtsustatud vaade menüüriba ja tekstiredaktoriga. Faili sisu on redaktoris koheselt kuvatud.

1.1. Menüüriba sisaldab esialgu vaid jooksutamise nuppu.

1.2. Klõpsates jooksutamise nupul asendub tekstiredaktor konsooliga, kuhu kuvatakse programmi väljund, ning menüüribale tekib konsooli peitmise/kuvamise nupp.

2. Lisatud kaustad ja failid peavad olema kuvatud failipuuna.

2.1. Laiendamata kaustal klikkides kuvatakse kausta otsesed lapsed.

2.2. Laiendatud kaustal klikkides peidetakse kausta otsesed lapsed.

2.3. Failil klõpsates avaneb tekstiredaktor, milles kuvatakse kasutajale faili sisu. See fail muutub aktiivseks ning kuvatakse avatud failide paneelil.

3. Avatud failide paneel kuvab avatud/aktiivseid faile.

3.1. Avatud failide paneelil failile klõpsates muudetakse see aktiivseks. Aktiivse faili sisu kuvatakse tekstiredaktoris.

3.2. Klikkides avatud failide nimekirjas oleva faili juurde kuuluvale jooksutamise nupule, käivitatakse serveris kõnealune fail. Programmi väljundi näitamiseks kuvatakse konsool, mis sisaldab väljundit teksti kujul.

3.3. Klõpsates avatud failide nimekirjas oleva faili juurde kuuluvale sulgemise nupule, eemaldatakse vastav fail avatud failide paneeli nimekirjast.

3.3.1. Kui avatud faile on lisaks suletavale veel, muudetakse aktiivseks suletud failile eelnev või järgnev fail. Vastav muutus kajastub ka tekstiredaktoris.

3.3.2. Kui suletav fail on viimane avatud fail, peidetakse tekstiredaktor.

4. Tekstiredaktor võimaldab kasutajal kuvatavat koodi muuta.

5. Konsool kuvab kasutajale programmi väljundit, aktiivse konsooli faili, selle faili jooksutamise nuppu ning valikut kõikidest avatud konsoolidest.

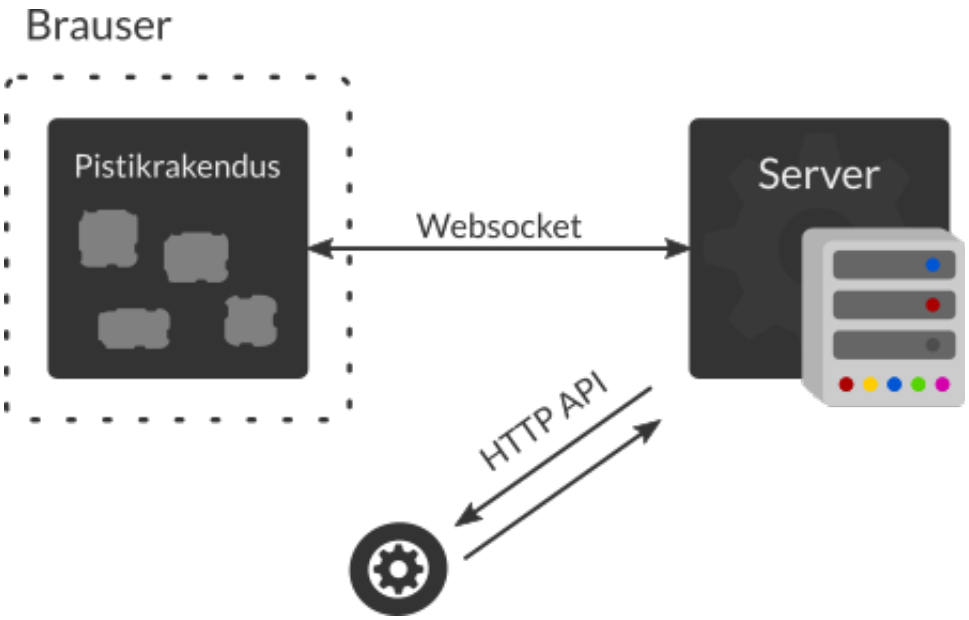
5.1. Interaktiivsete programmide puhul on kasutajal võimalik sellele konsooli kaudu sisendit saata.

2.3 Kõrgtaseme arhitektuur

Platvormi defineerivad kaks olulist komponenti: klient ja server. Tegemist on tüüpilise klient–server arhitektuuriga, kus kliendil on vajadus mingisuguste andmete järele ning serverirakendus peab neid vajadusi rahuldama.

Kuna tegemist on platvormiga, mille põhitegevus hõlmab koodi reaalajas jooksutamist, on oluline ka transpordikihis toimuv. Rakenduse ja serveri vaheline suhtlus toetub täielikult Websocketi[1] tehnoloogiale, mis on TCP-l põhinev täisdupleks kommunikatsioonikanal [4] . Platvormi kasutatavuse laiendamise eesmärgil on oluline osa ka HTTP API-l, mis käesoleva töö raames implementeeritakse eelkõige JOOP-ga liidestumise eesmärgil. Kuigi see on ka paljude teiste kasutusjuhtude jaoks sobilik meedium, seda muus kontekstis vaatluse alla ei võeta.

Joonisel 1 on graafiliselt kujutatud platvormi kõrgtaseme arhitektuur.



Joonis 1: Platvormi kõrgtaseme arhitektuur. (Autori joonis)

Edasi vaadeldakse kliendipoolse rakenduse ja serverilahenduse spetsiifikat eraldi.

[1] https://en.wikipedia.org/wiki/WebSocket

3 Kliendipoolne pistikrakendus

Töö üheks tulemiks on platvormi kasutajaliides rakenduse näol, mis võimaldab programmi lähtekoodi jooksutada ning näha selle väljundit. Rakendust saab kasutada selle veebilehte süstides. Kogu kood on avalik ning kättesaadav GitLabi keskkonnast[1].

3.1 Kasutatud tehnoloogiad

Rakendus luuakse kasutades kaasaegseid veebiarendamise tööriistu ja praktikaid. Järgnevalt on toodud olulisemad kasutajaliidese loomisel kasutatavad tehnoloogiad ning põhjendused/kirjeldused, miks neid kasutada on otsustatud:

- Node.js[2] – sündmusjuhitav Javascripti käituskeskkond, mis võimaldab luua brauseris kasutatavaid lahendusi npm[3]-st kättesaadavate teekide kaasabil [5] ;
- Vue.js[4] – modernne efektiivseid kasutajaliideseid luua võimaldav teek;
- Vuex[5] – Vue.js-ga kasutatav flux[6] arhitektuuri implementatsioon (tsentraliseeritud olekuhaldus) [7] ;
- Webpack[7] – Node.js-i moodulite pakkija;
- Babel[8] – Javascripti kompilaator, mis võimaldab kaasaegse Javascripti (ECMAScript 6) brauseris jooksutamist [6] ;
- Websocket – TCP-l põhinev täisdupleks kommunikatsioonikanal, mis võimaldab rakendusel ja serveril suhelda reaalajas [4] ;

1 https://gitlab.com/embeddabl/embeddabl
2 https://nodejs.org/en/
3 https://www.npmjs.com/ - Node.js-i paketihaldur
4 https://vuejs.org/
5 https://github.com/vuejs/vuex
6 https://facebook.github.io/flux/ - veebirakenduste kasutajaliideste ehitamiseks kasutatav disainimuster
7 https://webpack.github.io/
8 https://babeljs.io/

- ACE[1] – tekstiredaktor, mis toetab koodi süntaksi esiletõstmist jm programmeerimiseks vajalikke baasfunktsionaalsusi;
- SCSS[2] (Sass) – tavalise CSS-i edasiarendus, loob stiilidokumendile uue perspektiivi: võimaldab lauseid taaskasutada ja kogu dokumenti paremini hallata;
- HTML.

Järgmisena näidatakse, kuidas nende tehnoloogiate omavahelise sümbioosi tulemusena kliendipoolne rakendus on üles ehitatud.

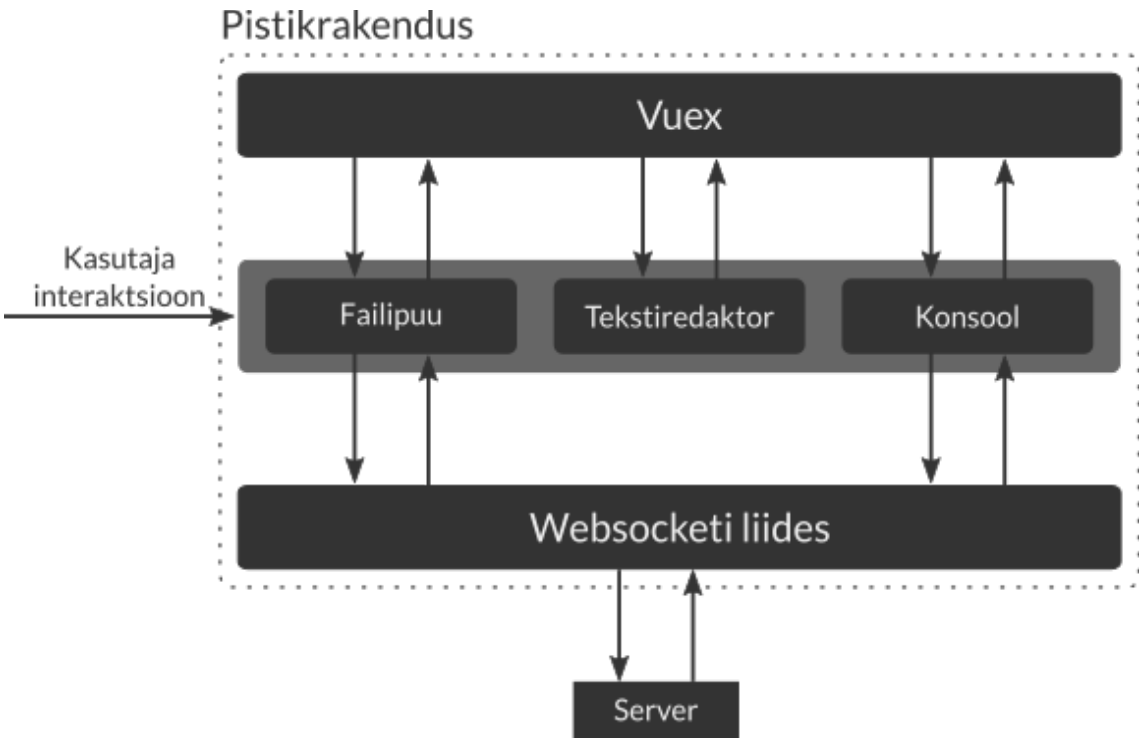
3.2 Realisatsioon

Rakenduse kasutajaliidese ning kõikide selle interaktsioonide loomiseks kasutatakse Vue.js-i, mis on kerge, kiire ja komponeeritav vahend modernsete liideste valmistamiseks. Efektiivsemaks sisemiste olekute haldamiseks kasutatakse Vuex-i. Brauseris rakenduse töölesaamiseks on vaja veel läbida üks samm: lähtekoodi transpileerimine[3]. Kogu kood ühildub ECMAScript 6 standardiga, mis on ühe uusima Javascripti versiooni spetsifikatsioon. Kuna see versioon ei ole praeguste brauserite poolt toetatud, tuleb see transformeerida neile arusaadavale kujule. Selleks kasutatakse Babeli transkompilaatorit, mis teeb just seda.

Joonis 2 kujutab pistikrakenduse põhielemente ja annab aimu, kuidas toimub andmete liikumine rakenduse sees, millised osad võtavad vastu kasutajapoolset sisendit ning kuidas toimub andmevahetus rakenduse komponentide ja serveri vahel.

Edasi kirjeldatakse rakenduse visuaalset, nähtavat liidest, sisemist ehitust ja seal toimuvaid protsesse.

1 https://ace.c9.io/
2 http://sass-lang.com/
3 https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/



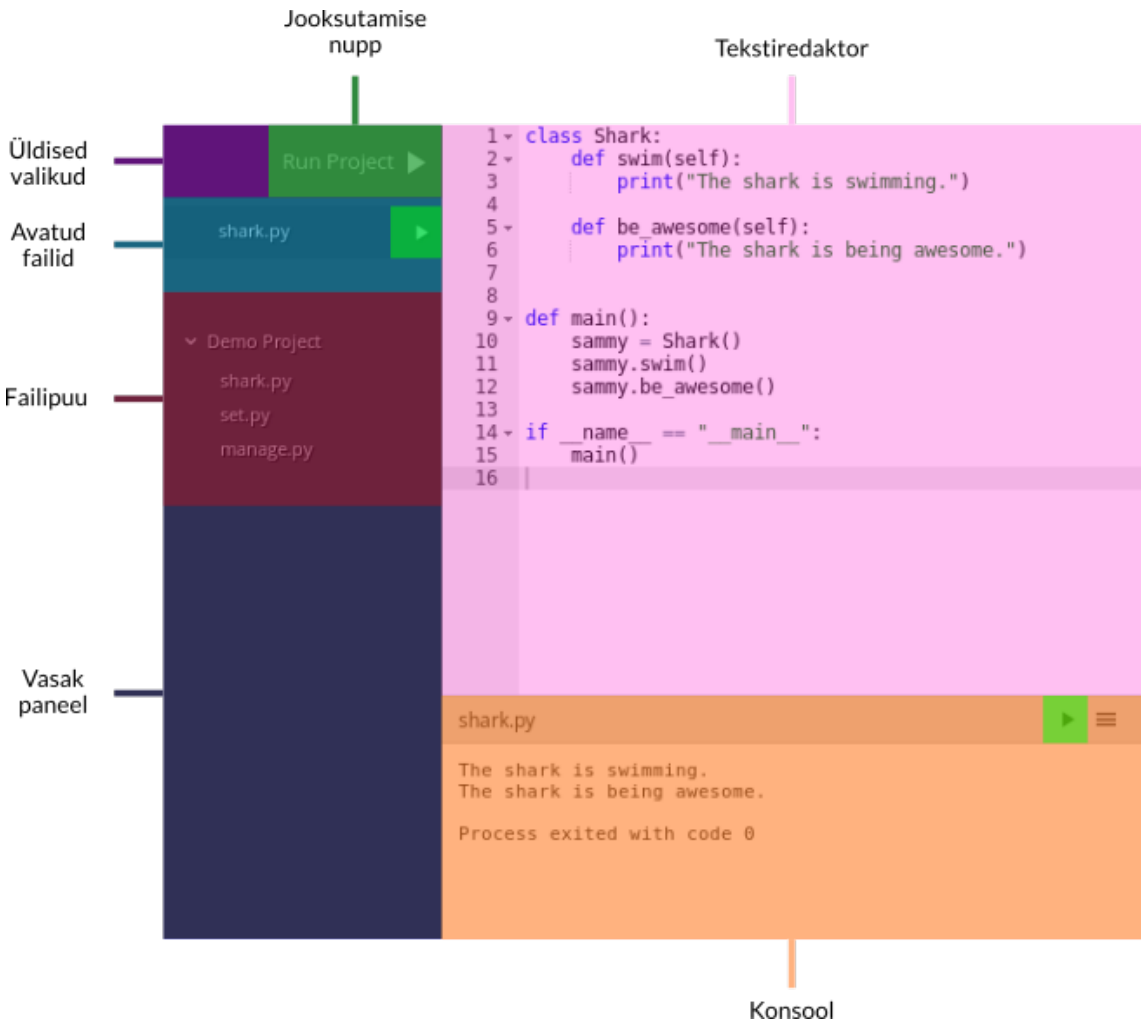
Joonis 2: Rakenduse põhielemendid ja andmete liikumine. (Autori joonis)

3.2.1 Kasutajaliidese ülesehitus

Vue.js-i ideoloogia seisneb komponendipõhises arhitektuuris, millest on saanud praeguse aja kliendipoolsete rakenduste arendamise de facto standard. Nii on rakendus tervikuna üks suur komponent, mis koosneb väiksematest tükikestest, mis võivad omakord sisaldada teisi tükke jne. Komponendid on küll sisemine abstraktsioon, kuid selle tulemus kajastub selgesti visuaalselt. Selline koodi ülesehitus muudab arendusprotsessi lihtsamaks ning hallatavamaks, sest komponendid on korduvkasutatavad ja aitavad rakenduse eri osi paremini abstraheerida. Samuti on igal elemendil oma ülesanne, mida ta täidab.

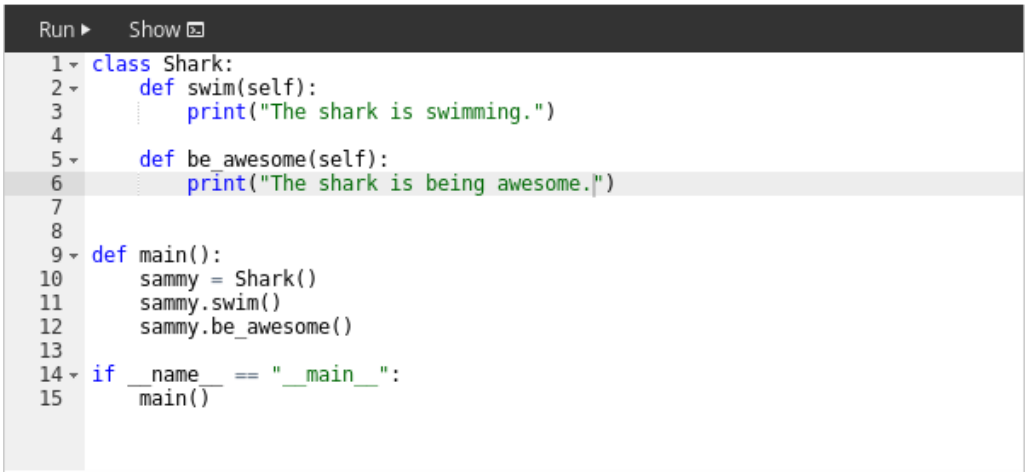
Joonis 3 kujutab kasutajaliidese komponente.

Liidese eri osad on interaktiivsed ning kasutaja tegevustele reageerivad: klikitavad elemendid muudavad värvi; failipuud laiendades kuvatakse kausta alamfailid ja -kaustad; failil klikkides kuvatakse selle sisu; interaktiivse programmi käivitamisel on võimalik sellele läbi konsooli sisendit saata jne.

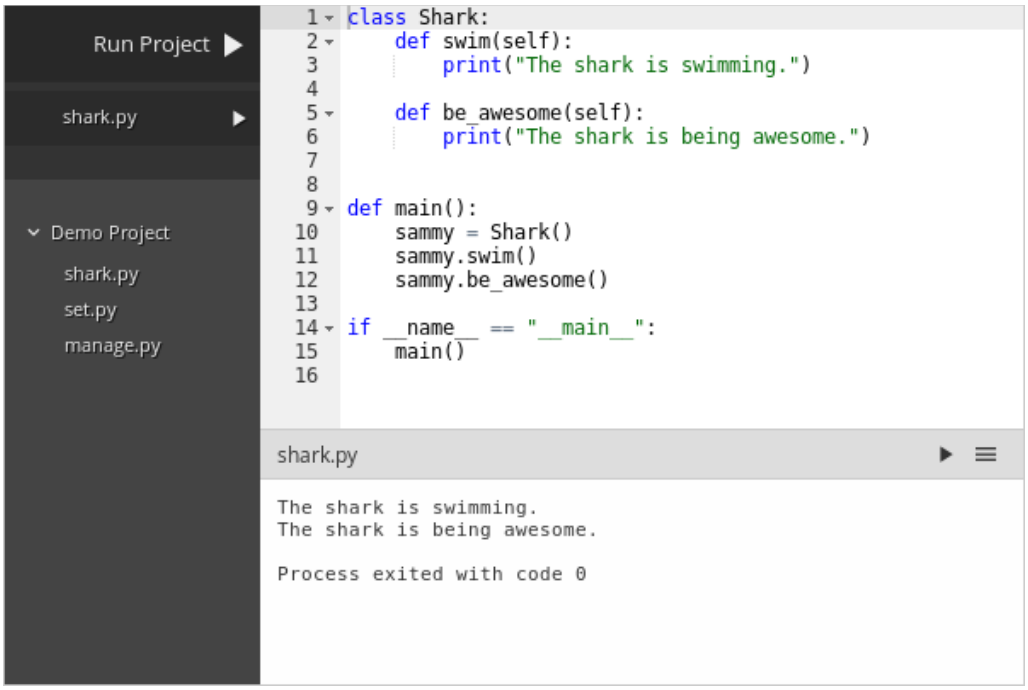


Joonis 3: Ülevaade rakenduses kasutatavatest Vue.js komponentidest. (Autori joonis)

Laiemas laastus on rakendus kasutajaliidese poolest kaheilmeline. Olenevalt, mitu faili sisse antakse, aktiveeritakse kas ühe või mitme faili vaade – kujutatud vastavalt joonistel 4 ja 5.



Joonis 4: Ühe faili vaade. (Autori joonis)



Joonis 5: Mitme faili vaade. (Autori joonis)

3.2.2 Andmete liikumine rakenduses

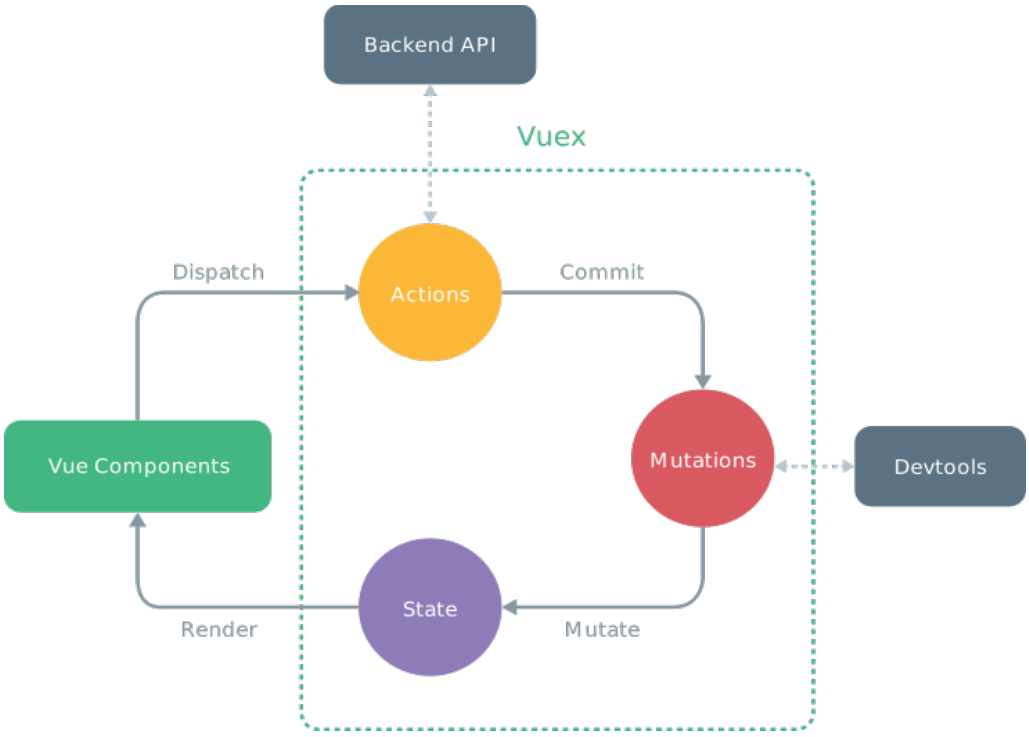
Esimese asjana, kui kasutaja rakenduse lehte süstib, laetakse CDN-ist (sisuedastusvõrgust) brauserisse ACE tekstiredaktor. Edasine kulg sõltub parameetritest, mille kasutaja süstides rakendusele sisse andis. Kui faile ei pea serverist laadima, on rakendus kasutamiseks valmis. Vastasel juhul saab seda kasutada alles pärast failide alla laadimist.

Nagu eelmises peatükis mainiti, on rakendus sisemiselt realiseeritud komponente kasutades. Kuigi kõikidel komponentidel on oma ülesanne, on osad neist siiski omavahel tihedalt seotud. Kogu rakenduse kulg saab alguse peakomponendist, mis loob endas olevad komponendid. Nii hargneb see protsess kuni komponendid enam ühtegi teist ei sisalda.

Komponente seovad lisaks veel andmed. Kuigi Vue.js implementeerib vaikimisi kahesuunalist sidumist (*two-way binding*), on seda sügava komponentide hierarhia puhul ebamugav kasutada. Seda seepärast, et mitu komponenti võivad omavahel jagada sama olekut ning vaikimisi pakutud meetod muudab koodi duplikeerimise tõttu raskesti hallatavaks. Selle probleemi vältimiseks on autor otsustanud rakendusesiseste olekute haldamiseks kasutada flux arhitektuuril põhinevat lahendust. Vue.js-i jaoks on selleks

Vuex, mis võtab andmed komponentidest välja ning tekitab tsentraalse olekuhaldamise mehhanismi, muutes koodi paremini hallatavaks. [7]

Joonis 6 illustreerib ühesuunalist andmete liikumist Vue.js-i komponentide ja Vuex-i vahel.



Joonis 6: Andmete liikumine Vue.js komponentide ja Vuex vahel. [8]

Vuex-i üksus haldab näiteks failide, konsoolide ja Websocketiga seonduvat. Need on põhilised elemendid, mis on mitmete erinevate komponentide poolt kasutuses: komponent kas muudab või sõltub selle elemendi olekust.

3.2.3 Reaalajas kommunikatsioon

Reaalajas suhtluse serveriga tagab Websocketi (WS) liides. Komponentide seest on võimalik Vuex-i kaudu küsida WS objekt ning läbi selle teha päringuid. WS objekti funktsionaalsust laiendab päriluse (*inheritance*) kaudu Node.js-i sisseehitatud klass *EventEmitter*, mis aitab serveriga suhtlemiseks luua väga paindliku lahenduse.

Nii on võimalik kuulata erinevaid kanaleid, mille kaudu andmed liiguvad. Joonisel 7 on näha, kuidas koodi jooksutamisel püüab konsooli komponent kinni serverist tulnud programmi väljundi.

```
this.getWs.on('programOutput-' + this.file.path,
this.onOutput.bind(this));
```

Joonis 7: *EventEmitteri* kasutamine programmi väljundi kuulamiseks.

Kuna andmete liikumine Websocketi täisdupleks kanalis pole kuidagi piiratud, on võimalik jooksutada mitut programmi samal ajal. Näiteks võib töös olla korraga 3 programmi, mis aktiivselt väljundit saadavad. See on võimalik, sest igale programmile on loodud omaette eraldatud kanal, mille kaudu nende jaoks loodud konsoolid sündmuseid kuulavad.

3.2.4 Failide laadimine

Eelnevalt kirjeldatud reaalajas kommunikatsioon on aluseks ka failide serverist laadimisele. Loodud rakendus toetab koodi laadimiseks erinevaid viise (täpsemalt peatükis 3.3.1). Kõikide projektitüüpide puhul – peale JSON-i – on vajalik faile kasutajale kuvamiseks esmalt serverist küsida. Võimalik oleks ka kõik failid korraga saata, kuid suurte projektide puhul pole see kohe kindlasti kõige mõistlikum tegu, sest nii peab suure hulga andmeid ühekorraga kliendile saatma.

Kasutajakogemuse sujuvamaks tegemiseks, et failide järele ootama ei peaks, rakendatakse nö laisa laadimise (*lazy loading*) disainikontseptsiooni. Selle idee seisneb asjaolus, et ressursse tuleks juurde küsida alles siis, kui selleks vajadus tekib [9] . Nii päritaksegi serverist faile alles siis, kui kasutaja failipuus kausta avada otsustab. Selline teguviis võib kõikide failide korraga laadimisega võrreldes rakenduse töövõimet oluliselt tõsta, sest alati ei lähe kõiki faile vaja. Samuti vähendab see serveri töökoormust, sest andmeid saadetakse vähem.

Laisa laadimise efektiivsust kasutajakogemuse vaatepunktist ei takista ka väike viivitus, mis failide saatmisele kulub. Keskmiselt võtab ühe kausta failide laadimine väga vähe aega, nii et kasutaja seda peaaegu ei märka.

3.3 Rakenduse kasutamine

Lisaks rakenduse visuaalsele disainile on väga olulisel kohal ka kasutajakogemuse (UX) disain. Lihtne on mingi asi suvaliselt valmis teha ilma disainiotsuseid sügavamalt läbi mõtlemata, kuid hiljem võivad need otsused saada määravaks, kas kasutajad hakkavad

või ei hakka loodud tarkvara kasutama. Käesolevas peatükis kirjeldatakse kahte kõige olulisemat kasutusjuhtu: rakenduse kasutamine seda süstiva kasutaja ning koodi jooksutava kasutaja poolt.

3.3.1 Veebilehte süstimine

Selleks, et rakendus veebis kasutajatele kättesaadavaks muuta, tuleb see kõigepealt lehekülje lähtekoodi sisestada. Siinkohal on mõistlik kasutada levinud meetodeid, millega kasutajad harjunud on. Kõige lihtsam oleks kasutada *iframe* elemendiga süstimist, mis põhimõtteliselt kujutab endast ühe veebilehe teise sisestamist. Teine meetod on teek kõigepealt brauserisse laadida ning seejärel luua uus objekt, mis rakenduse kasutajale kuvab. Töös realiseeritakse teine meetod, kuna see on seadistamise seisukohast palju paindlikum. Joonisel 8 on näidatud, kuidas giti repositooriumist koodi lehte süstimise protsess välja näeb.

```
<div id="embeddabl"></div>
<script src="embeddabl.js"></script>
<script>
  var embeddabl = new Embeddabl({
    el: "#embeddabl",
    project: {
      type: "GIT",
      name: "git-demo",
      url: "https://gitlab.com/embeddabl/git-demo.git"
    }
  });
</script>
```

Joonis 8: Giti repositooriumist koodi veebilehte süstimise protsess.

Süstitud rakenduse seadistamiseks on kasutajal võimalus tüübi konstruktorile ette anda erinevaid valikuid. Ülevaade valikutest on järgnev:

- *el* – määrab elemendi, millesse rakendus süstitakse;
- *project* – koondab projektiga seonduvad seaded – järgnevad seaded asuvad selles blokis;
- *type* – projekti tüüp, võimalikud valikud: GIT, ZIP, JSON, USER;

- *name* – projekti nimi, määrab kausta, kuhu sisse projekti failid pannakse või võetakse. Kohustuslik GIT, ZIP ja USER tüüpi projektide puhul;
- *url* – internetiaadress, kus projekt asub. Kohustuslik vaid GIT ja ZIP tüüpi projektide puhul;
- *files* – võimaldab faile lisada otse brauserist, peab olema JSON formaadis. Kohustuslik vaid JSON tüüpi projekti puhul;
- *user* – määrab kasutaja konteineri, millest projekt laetakse. Kohustuslik vaid USER tüüpi projekti puhul.

3.3.2 Koodi jooksutamine ja interaktsioon programmiga

Koodi jooksutamiseks peab rakendus olema eelnevalt seadistatud. Esimese sammuna tuleb soovitud fail avada ning seejärel käivitada vasakpoolsest menüüst avatud failide paneelist (kasutajaliidesest ülevaate saamiseks vt joonist 3).

Vajadusel on kasutajal võimalik lähtekoodi muuta. Selleks peab fail avatud ning aktiivne olema. Faili muutes märgitakse see „mustaks” ning jooksutades saadetakse kõik „mustad” failid serverisse. Kui failid on salvestatud, käivitatakse programm.

Interaktiivsetele programmidele, mis ootavad kasutajapoolset sisendit, loob autor ise lihtsa tekstiredaktori, mis puhverdab sisendit, kuni kasutaja vajutab *Enter* klahvi. Seejärel saadetakse kirjutatud tekst serverisse ning edastatakse programmi protsessile.

4 Server

Loodava platvormi tuum paikneb serveritehnoloogias. Käesolev peatükk annab esmalt ülevaate kasutatud tehnoloogiatest, seejärel täpsemalt olulisematest protsessidest, mis platvormi toimimiseks vajalikud on. Kood on kättesaadav GitLabist[1].

4.1 Kasutatud tehnoloogiad

Kuna tegemist on serverirakendusega, on autor selle loomise keskkonnaks valinud Ubuntu operatsioonisüsteemi eelkõige rikkaliku paketirepositooriumi ja olemasolevate tööriistade kättesaadavuse pärast. Lisaks on autoril eelnev Linuxi administreerimise kogemus.

Nimekiri kasutatud tehnoloogiatest ja nende valituks osutumise põhjustest:

1. Ubuntu 16.10[2] – ennast tõestanud serveri OS; rikkalik dokumentatsioon ja paketirepositoorium;

2. Node.js – rakenduse asünkroonse iseloomu pärast igati sobilik valik; keskkonna olemus võimaldab kiiret prototüüpimist ning sisaldab tohutul hulgal teeke;

3. Websocket – võimaldab rakenduse ja serveri vahel luua reaalajas kommunikatsioonikanali;

4. Python[3] – lihtsasti kasutatav kõrgtaseme keel;

5. LXC[4] – vahend turvaliste isoleeritud konteinerite loomiseks Linuxil [10] ;

[1] https://gitlab.com/embeddabl/server
[2] https://www.ubuntu.com/
[3] https://www.python.org/
[4] https://en.wikipedia.org/wiki/LXC

6. BTRFS[1] – failisüsteem, mis toetab *Copy-on-Write* tehnoloogiat, tehes sellest üliefektiivse failisüsteemi, ning võimaldab kasutajate kõvaketta kasutusele panna dünaamilise piirangu [11] ;

7. OverlayFS[2] – failisüsteem, mis võimaldab kasutajate konteinereid kloonida nii, et aluskonteineri muudatused kajastuvad kloonitud konteineris; samuti ei võta kloonitud konteinerid esialgu praktiliselt üldse kettamahtu: lisaruum kaasneb vaid selles konteineris loodud uute failide tekkimisel, mida aluskonteineris ei eksisteeri [12] ;

8. systemd[3] – Ubuntu 16.10 vaikimisi teenustehaldur, mis võimaldab lihtsasti luua ka enda kohandatud teenuseid;

9. Redis[4] – kiire ja kerge mälus hoitav andmebaas ja sõnumivahendamise meedium, mis võimaldab eri protsesside vahel sõnumeid saata [13] ;

10. Express[5] – kiire ja minimalistlik raamistik veebiliideste loomiseks [14] .

4.1.1 Kasutatava konteineritehnoloogia taust

Kui kasutaja platvormiga programmi jooksutamise eesmärgil ühendub, luuakse talle kõigepealt seda tegevust võimaldav konteiner ehk isoleeritud ruum, mis on baassüsteemist eraldiseisev keskkond. Konteineri seest paistab nagu oleks tegemist omaette eksisteeriva süsteemiga. Kõikide töös käsitletavate konteineritega seotud teemade aluseks on projekt nimega *Linux Containers* (lühidalt LXC), millega alustati 2008. aastal, kuid stabiilsus saavutati alles 2014. aastal.

LXC pakub konteinerite loomiseks laialdasi võimalusi, lastes kasutajal valida endale meelepärasem konfiguratsioon. Toetatud on konteineritest hetktõmmiste tegemine ja nende kopeerimine. Kuna konteinerid on tänapäevases tarkvara arendamise protsessis olulisel kohal, on toetatud ka erinevad turvalisusega seotud aspektid. Igale konteinerile on võimalik sätestada mälu, protsessori ja kõvaketta *I/O* piiranguid [10] . Nende

[1] https://en.wikipedia.org/wiki/Btrfs
[2] https://en.wikipedia.org/wiki/OverlayFS
[3] https://en.wikipedia.org/wiki/Systemd
[4] https://redis.io/
[5] https://expressjs.com/

parameetritega konteinerite konfigureerimine on suureks abiks *DoS*-rünnete tõkestamiseks.

Omaette turvalisuse teemaks on privileegideta konteinerid, mis takistavad pahatahtlikel kasutajatel serverile kahju tegemast [10] . LXC loojad on teadlikud mitmetest meetoditest, kuidas konteineri isolatsioonist välja murda ning baassüsteemile ligipääs saada [10] . Ka privileegideta konteineritest on võimalik välja pääseda, kuid kuna kasutajal, kes sel juhul baassüsteemini pääseb, pole seal ühtegi õigust, on see piisavalt turvaline, et tundmatutel kasutajatel platvormi kasutada lasta [10] . Kõik töös kasutatavad konteinerid on privileegideta.

Teiseks oluliseks aspektiks on failisüsteemid, mida LXC toetab. Käesoleva töö seisukohast on üliolulisel kohal OverlayFS, ilma milleta oleks konteinerite kasutamine praktiliselt mõttetu. Nimelt võimaldab OverlayFS luua väga kergeid konteinereid, mis oma mahult on vaid ~3KB. Seda kasutatakse koodi jooksutavate konteinerite loomisel. Lisaks on toetatud ka näiteks BTRFS, ZFS, LVM, kuid neid konteineritega seoses ei rakendata.

4.2 Realisatsioon

Oluline on, et loodav lahendus oleks hallatav ning vajadusel kergesti laiendatav. Lisaks peab süsteemi ülesehitus olema piisavalt modulaarne, järgides kõrge kokkukuuluvuse ja madala sidususe ideoloogiat.

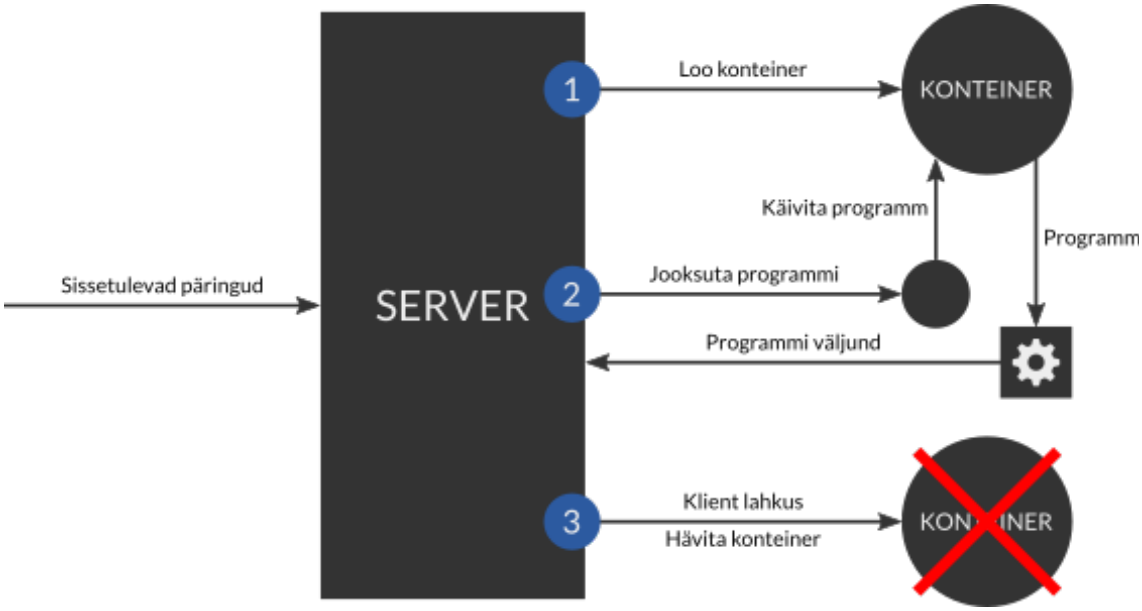
Käesoleva töö terviklik serverilahendus ei seisne ühes konkreetses programmis, vaid sisaldab mitmeid erinevaid omaette eksisteerivaid lülisid, mis aitavad täita üldist eesmärki: kasutaja soovitud koodi jooksutamist. Lahenduse olulisemad lülid on kahte – Websocketi ja HTTP – veebiliidest realiseerivad serverirakendused. Need liidesed kasutavad systemd haldusteenust, mis vajavad konteinerite loomise ettevalmistamiseks juurkasutaja õiguseid. Lisaks luuakse erinevaid abistavaid skripte, mis võimaldavad teatud toimingute automatiseerimist.

Kuigi lahendus sõltub paljudest erinevatest elementidest, on serveri tööpõhimõte siiski üsna lihtne. Joonisel 9 on toodud koodi jooksutava konteineri, kõige olulisema

käesolevat tööd defineeriva kontseptsiooni alustehnoloogia, üldine elutsükkel, mis jaguneb kolme faasi:

1. konteineri loomine;
2. programmi jooksutamine;
3. konteineri hävitamine.

Käesolevas peatükis vaadeldakse, kuidas toimib taustal toimuv tegevus igas kujutatud faasis spetsiifilisemalt (sisemine liidestus ning komponentidevaheline koostöö).



Joonis 9: Konteineri elutsükli kolm faasi. (Autori joonis)

4.2.1 Koodi jooksutava konteineri loomine

Isoleeritud ruumi loomine algab BTRFS failisüsteemi alamköite (*subvolume*) tekitamisega. Kõiki BTRFS käske täidab Pythoni skript, millest on tehtud systemd teenus, mis käivitatakse, kui server tööle pannakse. See skript saab infot veebiliidestelt, mis ütlevad toimingu, mida tegema peab. Toimingute edastamiseks kasutatakse *publish/subscribe* (*pub/sub*) mustrit, mis on Redises vaikimisi implementeeritud. Näiteks on üheks toiminguks *createContainer*, mis tähendab, et peab uue BTRFS alamköite looma. Selle saavutamiseks jooksutatakse skriptis käsku `btrfs sub create /<asukoht>/<alamköite_nimi>`. On oluline mainida, et kuna kõiki BTRFS-iga seonduvaid toiminguid tehakse juurkasutajana, peab loodud alamköite omaniku muutma serverirakendust jooksutavaks kasutajaks. Seda saab teha käsuga `chown <kasutaja>:<kasutaja> /<asukoht>/<alamköite_nimi>`.

Loodud BTRFS köitele määrati esialgu dünaamiline kettamahu limiit, millest kasutaja üle minna ei saanud – näiteks 60MB. Käsk selle saavutamiseks on `btrfs qgroup limit -e 60M /<asukoht>/<alamköite_nimi>`, kus *-e 60M* tähistab limiidi suurust megabaitides. Köitele limiidi määramisega tekkis aga autorile teadmatul põhjusel konteinerite loomisel ning koodi jooksutamisel viga, mis väitis, et kettamaht on täis ning tegevus katkestati. Lähemal uurimisel see aga nii ei olnud. Autor kahtlustab siinkohal BTRFS-i ja OverlayFS-i vahelist konflikti. Kuna konteineri kloonimisel *mount*'itakse alamkonteineri failisüsteem, mis on üle 1GB, siis BTRFS võib sellest segadusse sattuda ning seepärast arvata, et limiit on ammuilma lõhki. Sel põhjusel platvorm esialgu kettamahu piirangut sellisel kujul ei rakenda.

Järgmisena luuakse LXC konteiner, mis paigutatakse loodud BTRFS alamköite sisse. Konteiner tekitatakse käsuga `lxc-copy --name <aluskonteineri_nimi> --ephemeral --backingstorage overlay`, kus

- --*name* parameeter määrab konteineri, millest hetktõmmis tehakse;
- --*ephemeral* parameeter käivitab konteineri pärast loomist automaatselt ning määrab, et konteiner hävitatakse *lxc-stop* käsuga. Sisuliselt sisaldab see ka --snapshot parameetrit, mis tähendab, et aluskonteinerist ei tehta täielikku füüsilist koopiat, tänu millele nõuab loodav konteiner vaid ~3KB kettamahtu – ruumi võtavad vaid kasutaja loodud uued failid, mis aluskonteineris ei eksisteeri;
- --*backingstorage* on pandud *overlay*, mis katab aluskonteineri failisüsteemi nii, et aluskonteineris tehtud muudatused kajastuvad ka selle kloonides.

Loodud konteiner saab omale nimeks *<aluskonteineri_nimi>_XXXXXX*, kus X-ga on tähistatud LXC poolt genereeritud suvalised tähed ja numbrid.

Kuigi kõik konteineri loomiseks kasutatud parameetrid on olulised, on kõige olulisemaks siiski --*backingstorage overlay*. Mitte, et teised failisüsteemid halvad oleks, aga OverlayFS muudab implementeeritava platvormi palju paindlikumaks kui ülejäänud LXC poolt toetatud failisüsteemid.

Kinnitus konteineri loomise kohta saadetakse alles siis, kui selle võrguliidesed on tööle hakanud. See on vajalik, et jooksutataval programmil oleks ligipääs internetile.

4.2.2 Jooksutamiseks vajaliku koodi laadimine

Selleks, et kasutaja koodi jooksutada saaks, on pärast konteineri loomist vaja sellesse kood laadida. Olenevalt seadistusest laaditakse kood eri viisidel. Kui seadistuses on projekti tüübiks määratud

- GIT, laaditakse kood määratud giti repositooriumist;
- ZIP, laaditakse alla määratud ZIP fail, mis pakitakse lahti;
- JSON, peab olema määratud JSON noteeringus failipuu koos kaustade ja failide sisuga, mis konteinerisse salvestatakse;
- USER, kopeeritakse määratud kasutaja konteinerist kõik failid koodi jooksutavasse konteinerisse. Kopeerimise juures on oluliseks nüansiks asjaolu, et failidest ei tehta füüsilisi koopiaid ning need ei võta kõvakettal ruumi, kuni neid ei muudeta. Kuna kasutatakse BTRFS failisüsteemi ning see toetab *Copy-on-Write* tehnoloogiat, kopeeritakse kõik failid *reflinkidena*: `cp -a --reflink=always /<kasutaja_konteiner>/delta0 /<koodi_jooksutav_konteiner>`. *-a* parameeter määrab, et kopeerimisel jääksid alles originaalfailide õigused ja omanikud jne. d*elta0* kaust on OverlayFS failisüsteemi ülemine kiht, kus on kajastatud need muudatused, mille kasutaja on teinud. Seega tuleb vastav *delta0* kopeerida, et kasutaja failidele saaks koodi jooksutavast konteinerist ligi.

GIT, ZIP ja JSON tüüpi projektide korral teostatakse laadimine selleks loodud Pythoni skriptiga, mis on igast konteinerist kättesaadav. Laadimiseks kirjutatakse selle programmi standardsisendisse iga tüübi jaoks vajalik info, mille alusel skript eelnevalt defineeritud projekti tüübile vastavad toimingud sooritab.

4.2.3 Koodi jooksutamine

Eelnevalt seadistatud konteiner on valmis koodi jooksutama. Selleks on aga kõigepealt mõistlik defineerida metoodika, kuidas seda tehakse. Lahenduseks pakutakse välja bashi skriptid, mis sisaldavad standardseid protseduure programmikoodi kompileerimiseks ja/või jooksutamiseks. Lisaks annab see võimaluse koodihaldurile luua oma koodi jaoks

kohandatud skripte, mida platvorm ise vaikimisi ei paku. Vaikimisi skriptid on kättesaadavad ja ootavad kasutajate poolt täiendamist avalikus repositooriumis[1]. Töö raames luuakse kahe programmeerimiskeele jooksutamise skripti: *python3.run* ja *java.run*. Joonis 10 näitlikustab Java koodi jooksutamiseks vajalikku skripti.

```
#!/bin/bash
CLASSPATH=${3:-"."}
javac -cp "$CLASSPATH" $1
java -cp "$CLASSPATH" $2
```

Joonis 10: Java koodi jooksutamiseks kasutatav skript.

Kood käivitatakse kasutades Node.js-i sisseehitatud mooduli *child_process*[2] funktsioone *spawn* ja *exec*.

spawn funktsiooni kasutatakse koodi jooksutamisel kliendipoolsest rakendusest, kus pole teada, kui kaua programm töötab ning kas see küsib kasutajalt sisendit. Tekkinud väljund saadetakse kliendile tükikaupa. Iga tükk lisatakse rakenduses konsooli ning kuvatakse kasutajale.

exec funktsiooni kasutatakse HTTP API kaudu jooksutamisel. Kuna HTTP on ühenduseta protokoll, peab selle kasutamisel programm töö lõpetama ühe päringu jooksul, et väljund kätte saada. *exec* funktsioon puhverdab tekitatud väljundit ning programmi lõppedes saadetakse see koos võimalike vigadega kliendile päringu vastusena.

Joonisel 11 on toodud programmi käivitav funktsioon, mis pärast käivitamist registreerib ka väljundi tagasikutse (*callback*) funktsioonid. Nende funktsioonide kaudu jõuab väljund tagasi kliendini, kes jooksutamise päringu sooritas. Käsk, mille *_createCommand()* loob ning *spawn* funktsioonile sisse antakse on tavaline süsteemi käsk: `lxc-attach --name <konteineri_nimi> -- su <kasutaja> -c "cd ~ && <jooksutaja_skript> <jooksutaja_parameetrid>"`. Käsu selgitus:

1. *lxc-attach --name* <*konteineri_nimi*> käivitab konteineri sees määratud protsessi;

[1] https://gitlab.com/embeddabl/runners

[2] https://nodejs.org/api/child_process.html#child_process_child_process

2. *su* <*kasutaja*> muudab kasutajat, kes protsessi jooksutab. Vaikimisi on see juurkasutaja – turvalisem on seda teha mõne muu kasutajaga, kellel on vähem õiguseid;

3. *cd ~* muudab kasutaja kodukausta aktiivseks kaustaks;

4. <*jooksutaja_skript*> skript millega programmikoodi jooksutatakse. Vt näiteks joonist 10, kus on toodud näide skriptist, mis suudab Java koodi jooksutada;

5. <*jooksutaja_parameetrid*> parameetrid, mida skript korrektseks toimimiseks nõuab.

```
run() {
    let options = this._createCommand();
    this.process = spawn('lxc-attach', options);
    this.process.stdout
        .on('data', this._onstdout.bind(this));
    this.process.stderr
        .on('data', this._onstderr.bind(this));
    this.process.on('close', this._onclose.bind(this));
}
```

Joonis 11: Funktsiooni definitsioon, mis programmi käivitab.

4.2.4 Interaktiivsetele programmidele sisendi andmine

Programmid ei pruugi alati järjest tööd ära teha. On tavaline, et programmeerimiskeeled toetavad interaktiivset kasutajalt sisendi küsimist. Kuna see kasutusjuht on paljude kasutajate jaoks triviaalne nähtus, on oluline, et loodav platvorm seda ka toetaks. Rakendusepoolse kirjelduse selle kohta leiab peatükist 3.3.2.

Kui konteineris programm käivitatakse, aktiveeritakse iga programmi kohta kasutajapoolse sisendi kuulamise mehhanism. Sisendi registreerides juhitakse see konteineris jooksva protsessini. Kui programm sisendile reageerib, kasutatakse juba olemasolevaid tagasikutse funktsioone, mille kaudu väljund kasutajale saadetakse (joonis 11).

4.2.5 Koodi jooksutava konteineri hävitamine

Websocketi ja HTTP liideste puhul jõutakse konteinerite kustutamiseni erineval viisil.

Websocketi ühendus kujutab endast otspunktidevahelist kanalit, mille kaudu saavad mõlemad osapooled samaaegselt andmeid saata. Kanal hoitakse lahti senikaua, kui üks osapooltest otspunkti sulgeb. Kui sulgejaks on klient, siis registreerib server, et kasutaja lahkus ning kutsub välja funktsiooni selle konkreetse kliendi konteineri kustutamiseks.

Kuna HTTP liides on ühenduseta, alustatakse konteineri kustutamise protsessiga kohe pärast programmi töö lõppemist. Kustutamise kulg on mõlema liidese korral täpselt sama (vt joonist 12): peatatakse konteiner (tegemist on *ephemeral* tüüpi konteineriga, mis peatades kustutatakse) ning systemd konteinerite haldusteenus kustutab BTRFS alamköite. Mõistlik on lasta LXC-l endal konteiner ära kustutada, vastasel juhul jäävad alles ressursid, mis süsteemi tööd mõttetult koormama ja piirama hakkavad.

```
lxc-stop -n <konteineri_nimi> -k
btrfs sub delete <konteineri_alamköite_nimi>
```

Joonis 12: Konteineri kustutamisega seotud käsud.

4.2.6 HTTP liides

Platvormi HTTP liides luukse Express veebiraamistikku kasutades ning see realiseerib kaks otspunkti, mille ülesanded on kasutaja konteinerisse failide salvestamine ning koodi jooksutamine. See liides võimaldab erinevalt kliendipoolsest rakendusest käivitada koodi näiteks kasutajate enda serverist.

Ühe võimaliku kasutusjuhuna on see vajalik, kui koodi ei taheta avalikuks teha ning tulemusi on vaja enda tarbeks salvestada. Näiteks ülikoolide kursustes, kus on vaja tudengite koodi jooksutada. Käesolev liides võimaldab õppejõududel lisaks pistikrakendusele, mida saab õppematerjalides koodi interaktiivseks illustreerimiseks kasutada, kasutada sedasama platvormi tudengite koodi jooksutamiseks automatiseeritud moel. Taolisest võimalusest on huvitatud näiteks TTÜ kursuse „Objektorienteeritud programmeerimine Javas” läbiviijad.

Sarnaselt Websocketi liidesele kasutab ka HTTP liides kõiki eelnevalt kirjeldatud kolme faasi (vt joonist 9). Failide salvestamine toimub mõlema liidese puhul täpselt

samamoodi. Koodi jooksutamisel toob erinevuse sisse asjaolu, et väljundi saamiseks peab programm päringu jooksul töö lõpetama. HTTP puhul väljund puhverdatakse ning saadetakse programmi lõppedes päringu vastusena. Seega ei tohi programm küsida näiteks sisendit, vastasel juhul ühendus aegub.

Joonis 13 illustreerib HTTP liidese kasutamist *curl* programmi kasutades.

```
curl -d '{"type": "GIT", "name": "git-demo", "url":
"https://gitlab.com/embeddabl/git-demo", "user": "JOOP"}' -H
"Content-Type: application/json" -X POST
http://api.embeddabl.com/save-files
curl -d '{"type": "ZIP", "name": "git-demo", "url":
"https://gitlab.com/embeddabl/git-demo/repository/archive.zip",
"user": "JOOP"}' -H "Content-Type: application/json" -X POST
http://api.embeddabl.com/save-files
curl -d '{"file": {"path": "git-demo/demo.py"}, "user": "JOOP"}'
-H "Content-Type: application/json" -X POST
http://api.embeddabl.com/run
```

Joonis 13: HTTP liidese kasutamine failide salvestamiseks ja jooksutamiseks.

5 Loodud platvormi analüüs

Töö käigus valmis eelnevalt kirjeldatud tehnoloogiaid ja tehnikaid kasutades platvormi prototüüp. Käesolevas peatükis analüüsitakse loodud keskkonna koormuse all vastupidamist ning tehakse ettepanekuid, kuidas vastupidavust suurendada.

5.1 Koormuse testimine

Koormust testitakse loodud HTTP API kaudu, mis avab ühe otspunktiga ligipääsu platvormi „südamesse” ehk tehakse läbi kõik olulisemad protsessid, mis koodi jooksutamist hõlmavad.

Testimiseks kasutatakse Node.js-i keskkonnast kättesaadavat koormuse testimise tööriista nimega Artillery[1]. Lisaks HTTP liidestele võimaldab see testida ka Websocketi liideseid. Artilleryt saab seadistada konfiguratsioonifailiga, mis võivad olla kas YAML või JSON noteeringus. Nende seadistuste abil saab määrata testide käitumise. Võimalik on luua väga dünaamilisi teste, mis suudavad kasutajate käitumist veebilehel päris hästi jäljendada, kuid käesoleval juhul piisab ühest üsnagi lihtsast testist.

Joonisel 14 on toodud kasutatud testi konfiguratsioon, kus

- *target* on testitava liidese aadress;
- *phases* määrab erinevad faasid. Autor testis ühe faasina ning määras selle pikkuseks 30 sekundit. Iga sekundi järel lisandub 2 kasutajat;
- *scenarios* väli sisaldab kõikvõimalikke stsenaariume, mida testitakse. Võimalik on luua väga detailseid ja keerulisi stsenaariume, kuid siinkohal on vaid üks väga selge asi, mida testitakse. Loodud on üks stsenaarium, mis sisaldab POST päringu tegemist otspunktile */api/run* koos jooksutamise infoga JSON

[1] https://artillery.io/

noteeringus. Jooksutatakse faili *asd.py*, mis asub kasutajale *JOOP* kuuluvas projektis *PR01*.

```
config:
  target: "http://dev.embeddabl.com:8888"
  phases:
    - duration: 30
      arrivalRate: 2
scenarios:
  - flow:
    - post:
        url: "/api/run"
        json:
          file:
            path: "PR01/program.py"
          user: "JOOP"
```

Joonis 14: Artillery konfiguratsioon platvormi koormuse testimiseks.

Testimiseks kasutatud serveri tehnilised näitajad:

- Operatsioonisüsteem: Ubuntu 16.10
- Muutmälu (RAM): 1 GB
- Protsessor (CPU): Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz, 1 tuum

5.1.1 Testi tulemused

Pärast testi jooksutamist näitab Artillery, kuidas see õnnestus. Tulemused kajastuvad joonisel 15.

Kokku tehti 60 koodi jooksutamise päringut. Kõik päringud said küll serverilt vastuse, kuid üks neist tagastati veakoodiga 500, mis tähendab, et serveris tekkis veaolukord. Ülejäänud päringud täideti edukalt. Minimaalselt võttis üks päring aega 19 sekundit ja maksimaalselt 78 sekundit. Kuna päringud venisid pikaks, oli maksimaalselt vastust ootamas 57 klienti. Kui autor kasutajate hulka või testi pikkust suurendas, hakkasid ühendused aeguma: Artillery vaikimisi *timeout* on seadistatud 120 sekundile. Optimaalne päringu täitmise aeg peaks jääma nelja sekundi juurde.

```
Scenarios launched:  60
Scenarios completed: 60
Requests completed:  60
RPS sent: 0.58
Request latency:
  min: 19085
  max: 78887.6
  median: 71206.4
  p95: 76040.9
  p99: 78659.7
Scenario duration:
  min: 19092.4
  max: 78888.9
  median: 71209.2
  p95: 76042.2
  p99: 78661.2
Scenario counts:
  0: 60 (100%)
Codes:
  200: 59
  500: 1
```

Joonis 15: Koormustesti tulemused.

5.1.2 Jõudluse analüüs

Koormustestist selgus, et mingi mehhanism süsteemis mõjutab väga tugevalt platvormi koormuse all töötamise interaktiivsust. Järgnevalt uuris autor, mis tegur seda põhjustab.

Sooritades veel mõned koormustestid samal ajal süsteemi ressursside hõivatust *top*[1] ja *collectl*[2] programmidega jälgides sai selgeks, et põhisüüdlasi on kaks:

1. Kuna iga jooksutamise jaoks luuakse uus konteiner, mis hõlmab failide kopeerimist ja muud seadistamist, kasutab LXC agressiivselt *rsync*[3] käsku, mis kokku hõivas üsna arvestatava osa süsteemi protsessori töömahust;

2. Käivitades konteineris testis toodud programmi, hõivab konteiner 25MB RAMi. 57 samaaegse käivitamise korral on selleks numbriks *25*57=1425MB*, mis tähendab, et mälu sai otsa. Kuna muutmälu on süsteemil kokku vaid 1GB

[1] https://linux.die.net/man/1/top
[2] https://linux.die.net/man/1/collectl
[3] https://linux.die.net/man/1/rsync

(millest tegelikult vaba on vaid ~600MB), võttis operatsioonisüsteem appi *swap* ruumi, mis on muutmäluga võrreldes palju aeglasem [15] . Lisaks kasutab *swap* samuti väga intensiivselt protsessorit, nii et süsteem muutub kasutamatuks.

Lisades masinale ühe gigabaidi muutmälu juurde, erines tulemus esimese testiga võrreldes selle poolest, et kõige kiiremini vastati päringule 6.5 sekundiga (versus esimese testi 19s). Kõige aeglasem päring kestis 76 sekundit (versus 78s) – selle koha pealt suurt edasiminekut ei toimunud.

Katsetades, mitu koodi käivitamist on vaja teha, et protsessor täielikult hõivatud oleks, selgus, et piisab vaid kahest samaaegsest jooksutamisest. Konteinerite loomine on protsessori jaoks liialt kulukas, et iga kord uus konteiner tekitada. Samas ajalises mõttes võttis esimene käivitamine 4.5 ja teine 7.0 sekundit.

Kokkuvõtlikult võib järeldada, et ühelt poolt on käesoleva töö „pudelikaelaks” üks süsteemi tähtsam komponent: LXC. Ühe efektiivsuse tõstmise võimalusena on võimalik välja mõelda meetod, kuidas koodi jooksutavaid konteinereid taaskasutada, et neid iga kord uuesti looma ei peaks. Lisaks on võimalik eelnevalt luua mingi arv konteinereid, mida kohe kasutama saab hakata. Samuti on mõistlik uurida teiste sarnaste baastehnoloogiate kohta ning võimalusi nende omavahel kombineerimiseks.

Teiselt poolt tuleb mängu realiseeritava platvormi olemus ja kasutusjuhtude keeruline iseloom. Kuna jooksutamisi võib korraga olla väga palju, on üleüldises mõttes oluline, et süsteemil oleks piisavalt palju ressursse eelkõige protsessori, RAMi ja kettamahu näol, et kõikide jooksutamistega hakkama saada. Kui ressursside koha pealt saabub füüsiline piir, tasub mõelda horisontaalse skaleerimise ehk lisaserverite loomise peale ning koormus nende kõigi vahel võrdselt ära jagada.

6 Kokkuvõte

Käesoleva töö eesmärk oli eelkõige luua platvorm, mis võimaldab ühelt poolt kasutajatel programmide lähtekoodi interaktiivsemal moel jagada ning teiselt poolt jagatavat koodi intuitiivselt jooksutada.

Eesmärkide saavutamiseks võeti esimeseks sihiks veebilehte süstitava pistikrakenduse loomine, mis võimaldab koodi reaalajas jooksutada, seda modifitseerida ning muid triviaalseid programmeerimisega seotud tegevusi. Teiseks võeti sihiks serverilahenduse loomine, mis suudab kõiki pistikprogrammi toimimiseks defineeritud nõudeid rahuldada. Lisaks võeti eesmärgiks luua HTTP liides, et seda saaks kasutada TTÜ kursuse „Objektorienteeritud programmeerimine Javas” raames.

Töö käigus valmis eesmärkidest lähtuv platvorm, mille jaoks defineeriti spetsiifilised nõuded. Nende alusel realiseeriti vajalikud süsteemikomponendid modernsete *front-end* ja Linuxi keskkonna tehnoloogiate ja praktikate abil. Kliendipoolne rakendus loodi Vue.js-iga ning see vastab ECMAScript 6 standardile. Rakendus on lihtsasti veebilehte paigaldatav ning selle kasutajaliidese struktuur kasutajatele esmakokkupuutel kasutamiseks piisavalt selge. Serverilahendustena loodi Node.js-iga Websocketi ja HTTP liidesed, mille töö põhineb isoleeritud LXC konteinerite loomisel ning nendes kasutajate määratud koodi jooksutamisel.

Autor testis valminud platvormi vastupidavust koormustestidega ning leidis, et selle efektiivseks toimimiseks on vaja palju süsteemi ressursse. Kuna jõudlus polnud töö omaette eesmärk, pakuti välja võimalusi väljatöötatud süsteemi edasiarendamiseks ja uurimiseks kas alternatiivsete baastehnoloogiate kasutamise või tõhusama konteinerite rakendamise vallas.

Püstitatud eesmärgid said täidetud.

Valminud platvormiga on võimalik tutvuda aadressil http://demo.embeddabl.com.

Kasutatud kirjandus

[1] Copy-on-write. [WWW] https://en.wikipedia.org/wiki/Copy-on-write (13.05.2017)

[2] GIF. [WWW] https://et.wikipedia.org/wiki/GIF (17.05.2017)

[3] Potter, P. Süsteemi nõuete esiletoomine ja analüüs. [WWW] http://maurus.ttu.ee/sts/wp-content/uploads/2012/09/S%C3%BCsteemi-n%C3%B5uete-esiletoomine-ja-anal%C3%BC%C3%BCs.pdf (26.04.2017)

[4] Websocket. [WWW] https://en.wikipedia.org/wiki/WebSocket (28.04.2017)

[5] Node.js. [WWW] https://nodejs.org/en/ (08.05.2017)

[6] Babel. [WWW] https://babeljs.io/ (08.05.2017)

[7] What is Vuex? [WWW] https://vuex.vuejs.org/en/intro.html (09.05.2017)

[8] Vuex. [WWW] https://github.com/vuejs/vuex (10.05.2017)

[9] Lazy loading. [WWW] https://en.wikipedia.org/wiki/Lazy_loading (20.05.2017)

[10] LXC - Security. [WWW] https://linuxcontainers.org/lxc/security/ (10.05.2017)

[11] btrfs Wiki. [WWW] https://btrfs.wiki.kernel.org/index.php/Main_Page (10.05.2017)

[12] Brown, N. Overlay filesystem. [WWW] https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt (10.05.2017)

[13] Redis. [WWW] https://redis.io/ (10.05.2017)

[14] Express. [WWW] https://expressjs.com/ (12.05.2017)

[15] What is Swap Space? [WWW] https://www.centos.org/docs/5/html/5.2/Deployment_Guide/s1-swap-what-is.html (12.05.2017)