Tartu Ülikool
Arvutiteaduse instituut
Informaatika õppekava

Joosep Orasmäe

Logimisraamistiku ning veebirakenduse väljatöötamine programmeerimiskeeles Rust

Bakalaureusetöö (9 EAP)

Juhendajad:
Varmo Vene, Vambola Leping

Tartu 2024

Logimisraamistiku ning veebirakenduse väljatöötamine programmeerimiskeeles Rust

Lühikokkuvõte:

Tänapäeval liigub üha rohkem inimesi tellimuspõhiste teenuste juurest ise hallatavate alternatiivide juurde. See toob endaga kaasa aga suurel hulgal logifaile, mille haldamine võib tavakasutajale osutuda töömahukaks või ei tehta seda üldse. Selle lahendamiseks loodi programmeerimiskeeles Rust tavakasutajatele suunatud veebirakendus logifailide haldamiseks. Rust programmeerimiskeel on üks kiiremini kasvavaid programmeerimiskeeli, mis on tuntud oma turvalise mälu mudeli poolest. Lisaks loodi ka logimisraamistik, mis võimaldab kasutajatel võimalikult vähese vaevaga logida oma Rusti programmides toimuvat.

Võtmesõnad:

Rust programmeerimiskeel, veebirakendus, logifailid, logimisraamistik

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Developing a logging framework and a web app using the Rust programming language

Abstract:

More and more people are moving away from subscription based services to selfhosted alternatives. This brings with it a lot of log files, which the average user might not have the time to deal with. To address this problem, a web app was created using the Rust programming language which enables regular users to manage all of their log files from one place. The Rust programming language is one of the fastest growing programming languages in the world, known for its safe memory model. In addition to the web app, a logging framework was also created for Rust, which allows the users to easily log events in their Rust programs.

Keywords:

Rust programming language, web app, log files, logging framework

CERCS:

P170 Computer science, numerical analysis, systems, control

Sisukord

Sissejuhatus

Tänapäeval on üha sagedasemaks muutunud tellimuspõhine ärimudel, kus teenuse või tarkvara kasutamise õiguse eest tuleb maksta igakuist tasu. See on aga endaga kaasa toonud olukorra, kus inimesed soovivad võtta kasutusele hoopis avatud lähtekoodiga alternatiive ja neid ise hallata (ingl *selfhost*). Sellega kaasneb suurel hulgal logifaile, milles navigeerumine võib osutuda üsnagi ajamahukaks. Eksisteerivad olemasolevad lahendused ettevõtetele ja suurte andmemahtudega klientidele, aga tavakasutajatele suunatud lahendused puuduvad või on piiratud funktsionaalsusega. Selle lahendamiseks loodi Rust programmeerimiskeele abil uus, tavakasutajatele suunatud veebirakendus logide haldamiseks.

Rust on kujunenud üheks viimaste aastate kiiremini kavavaks programmeerimiskeeleks. Rust võimaldab programmeerijatel kirjutada kiireid, turvalisi ja usaldusväärseid programme, pakkudes samaaegselt head arendajakogemust. Juhtivad tehnoloogia firmad nagu Google ja Microsoft on järk-järgult suurendamas Rusti osakaalu oma koodibaasides, seda just parema mälu turvalisuse tõttu võrreldes programmeerimiskeeltega nagu C ja C++ [1]. USA Valge Maja küberturvalisuse direktor kutsus üles 26.02.2024 avaldatud pressiteates kiirendama mälu turvaliste keelte nagu Rust kasutuselevõttu, et vähendada küberrünnakute ohtu [2].

Töö eesmärgiks on luua veebirakendus logide haldamiseks ja logimisraamistik Rusti programmidele, mis võimaldaks kasutajatel võimalikult vähese vaevaga logida programmides toimuvat. Varasemalt eksisteerivad lahendused on liialt ressursi nõudlikud, keerulise seadistusprotsessiga või ei eksisteeri neid üldse.

Töö on jaotatud kolmeks peatükiks. Esimeses peatükis antakse ülevaade Rust programmeerimiskeelest. Teises peatükis kirjeldatakse Rusti logimisraamistiku loomise protsessi ja arendamise käigus tehtud valikuid. Kolmandas peatükis kirjeldatakse veebirakenduse loomise protsessi, antakse rakenduse arhitektuuriline ülevaade ning teostatakse rakenduse jõudlustestimine.

Mõisted ja terminid

Viit (ingl *pointer*) on andmetüüp, mis sisaldab endas mingi teise objekti mäluaadressi.

Skoop (ingl *scope*) on programmikoodi osa, milles on kindlad väärtused ja muutujad kehtivad.

Viidatüüpi muutuja eluaeg (ingl *lifetime*) on skoop, mille jooksul on antud viidas salvestatud mäluaadress kehtiv.

Tarkvarateek (ingl *library*) on tarkvara kogum, mis on mõeldud korduvaks kasutamiseks teiste programmide poolt.

Logi on andmefail, mis hoiustab programmi käituse käigus toimunud sündmusi kronoloogilises järjestuses.

Logikirje on üksik sissekanne logifailis.

Andmejooks (ingl *data race*) on olukord, kus ühele mälu aadressile kirjutab ja loeb korraga mitu programmi lõime.

Muteks (ingl *mutex*) on lukustusvahend, mis reguleerib ressurssi jagamist eri programmiharude vahel ja tagab kindluse andmejooksude vastu.

Tagakomponent (ingl *back-end*) on kasutaja jaoks peidetud süsteemi osa, mis täidab kasutaja käske ning töötleb andmeid.

Kliendipoolne komponent (ingl *front-end*) on süsteemi osa, mis pakub kasutajaliidest süsteemi tagakomponendiga suhtlemiseks.

Paisketabel (ingl *hashmap*) on andmestruktuur, mis salvestab andmeid võtme ja väärtuse paaridena.

Regulaaravaldis (ingl *regular expression*) on määratletud tähistuse ja ehitusega avaldis otsitava teksti malli ja käsitlusviisi spetsifitseerimiseks.

1. Rust

Rust on 2006. a Mozilla töötaja Graydon Hoare-i loodud üldotstarbeline (ingl *general-purpose*) programmeerimiskeel, mis lubab C keelte perekonnaga sarnast jõudlust parema süntaksi ja turvalisusega [3]. Populaarse programmeerimise teemalise foorumi StackOverflow iga aastases arendajatele suunatud küsitluses on Rust olnud aastatel 2016-2023 kõige armastatuim programmeerimiskeel ning alates 2022. a ka osa Linux-i operatsioonisüsteemi kernelist [4].

Rusti eristab teistest programmeerimiskeeltest ainulaadne mälumudel. Perkel JM kirjutab, et keeled nagu C ja C++ on võtnud „abiratasteta" lähenemise ning selle tulemusena on kuni 70% Microsoft-i poolt iga aastaselt parandatavatest tarkvara vigadest seotud just mäluga ning selle haldusega [3]. Rust seevastu on oma mälu reeglitega aga märgatavalt rangem, keeldudes kompileerimast programme, mis neid rikuvad [5]. See annab arendajatele kindluse, et kompileeritud programmid on mäluga seotud vigadest priid.

1.1 Andmetüübid ja muutujad

Rust on staatiliselt ja tugevalt tüübitud keel, mis tähendab, et kõik muutujate tüübid peavad olema teada kompileerimise aegselt [5]. Muutuja defineerimiseks kasutatakse võtmesõna **`let`** ning muudetava (ingl *mutable*) muutuja defineerimiseks tuleb lisada veel võtmesõna **`mut`**. Muutujat saab defineerida ka konstantsena kasutades võtmesõna **`const`**. Konstandid on staatilise eluajaga (ingl *lifetime*) ehk kättesaadavad kogu programmi töö vältel [6]. Joonisel 1 on välja toodud erinevate muutujate defineerimise näited.

```
let a = 1; // Tüübituvastus määrab a tüübiks vaikimisi i32
(märgiga 32-bitine täisarv)
let b; // Muutuja b defineerimine
b = 1; // Muutuja b initsialiseerimine.
let c: u32 = 1; // Määrame muutuja tüübiks hoopis u32 (märgita
32-bitine täisarv)
let mut d = 1; // d on võtmesõna mut abiga tehtud muudetavaks
d += 2; // Liidame arvu 2, d uueks väärtuseks saab 3
const E: i32 = 1; // Loome konstandi
```

Joonis 1. Muutujate defineerimise näited.

1.2 Viidatüüpi muutujate eluajad

Üks suurimaid mäluhaldusega seotud vigasid C perekonna programmeerimiskeelte puhul on järelkasutusviga (ingl *use after free*), kus viidatüüpi muutuja poolt talletatavalt aadressilt üritatakse infot lugeda peale selle vabastamist [5]. Sellise operatsiooni tulemus on defineerimata, tulemuseks võib olla programmi krahh või halvemal juhul toimib näiliselt kõik õigesti.

Rusti programmides pole järelkasutusvea esinemine võimalik, kuna kompilaator kontrollib viitade korrektset kasutust. Selleks on igal viidatüüpi muutujal kindel eluaeg, mille jooksul on garanteeritud, et viidas salvestatud mäluaadress on kehtiv. Lõppenud eluajaga viidast lugemise tulemuseks on kompileerimiseaegne viga [7].

Viidatüüpi muutujate eluaegade väljendamiseks on Rustis spetsiaalne süntaks. Joonisel 2 on toodud eluaegu kasutava funktsiooni signatuuri näide Rust programmerimiskeele ametlikust veebiõpikust. Eluaja parameeter on `'a` ja `&'a str` viit sõnelõikele (ingl *string slice*), mille eluaeg on vähemalt sama pikk kui eluaeg `'a` [7]. Lihtsustatult öeldes tagab parameetri `'a` kasutus selle, et nii `x` kui ka `y` poolt viidatavad väärtused elavad vähemalt sama kaua kui funktsiooni poolt tagastatav viide.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

Joonis 2. Näide eluaegu kasutava funktsiooni signatuurist [7].

Eluaegade korrektne kasutamine ning nendest arusaamine on Rusti üks keerulisemaid komponente, aga samas ka üks peamiseid selle keele turvalisuse tagajaid.

1.3 Väärtuste omamine (ingl *ownership*)

Rusti kõige omapärasemaks komponendiks võib lugeda väärtuste omamise süsteemi. Väärtuste omamisega on seotud kolm reeglit:

- Igal väärtusel on omanik;
- Igal väärtusel saab mistahes ajahetkel olla vaid üks omanik;
- Väärtuse omaniku eluaja lõppedes väärtus kustutatakse (ingl *drop*) [7]. See tagab, et väärtuse omaniku skoobi lõppedes vabastatakse alati väärtuse poolt hõivatud mälu ja seda ainult üks kord vältimaks topeltvabastust (ing *double free*).

Kompilaatori osa, mis kontrollib muutujate ja viitadega seotud omamisreegleid, nimetatakse laenuhalduriks (ingl *borrow checker*) [7].

Väärtuste omamisega on seotud kolm põhilist operatsiooni: liigutamine (ingl *move*), laenamine (ingl *borrow*) ja muudetav laenamine (ingl *mutable borrow*) [5]. Esimene neist muudab väärtuse omanikku ning teised kaks lubavad antud väärtust kasutada ilma seda omamata. Tavalise laenamise ja muudetava laenamise vahe seisneb selles, et esimese puhul pole laenajal võimalust laenatavat väärtust muuta, teise puhul aga on see võimalik. Laenamise teeb veel eriliseks ka see, et väärtuse omanik kaotab laenamise ajaks õiguse väärtust muuta, vältimaks olukorda, kus laenatud väärtus muutub ootamatult selle kasutamise ajal [3]. Järgnevalt vaatame neid kolme operatsiooni eraldi.

Väärtuse liigutamiseks, ehk omaniku vahetamiseks, tuleb väärtus teise muutujaga siduda. Joonisel 3 toodud näites üritatakse kasutada muutujat peale selle väärtuse liigutamist, mis pole lubatud. Oluline on teha vahet viidatüüpi ja arvtüüpi väärtustel. Antud näide toimiks vigadeta kui oleks kasutatud sõne asemel arvu, kuna Rust kopeerib vaikimisi arvtüüpe, mitte ei liiguta neid. Selle tulemusena oleks nii `a` kui ka `b` sisaldanud sama, teineteisest sõltumatut arvu.

```
let a = String::from("näide"); // Muutuja a omab sõnet
let b = a; // Nüüd omab muutaja b sõnet
println!("{}", a); // VIGA! Muutuja a kasutamine peale selle väärtuse
liigutamist.
```

Joonis 3. Näide väärtuse liigutamisest.

Väärtuse laenamiseks tuleb kasutada & operaatorit. Joonisel 4 on toodud näide sõne laenavast ning seda väljastavast funktsioonist. Kui funktsiooni signatuuris oleks `&str` asemel `String`, siis liigutataks selle funktsiooni väljakutsumisel kasutatav argument funktsiooni skoopi ja peale funktsiooni lõpetamist vabastataks sõne poolt kasutatav mälu. Laenamisega seda vabastust aga ei toimu ja sõnet saab ka peale funktsiooni lõppu edasi kasutada.

```
fn valjasta(a: &str) {
      println!("{}", a);
}
```

Joonis 4. Näide sõnet laenavast funktsioonist.

Üheks tavalise laenamise eeliseks muudetava laenamise ees on võimalus ühte väärtust samaaegselt välja laenata mitu korda, kuna on tagatud, et seda väärtust ei saa ootamatult muuta [7].

Väärtuse muudetavaks laenamiseks tuleb kasutada **`&mut`** operaatorit. Joonisel 5 on toodud näide sõnet muudetavalt laenavast ning seda muutvast funktsioonist. Muudetavalt laenatud väärtusega on lubatud sooritada pea kõiki operatsioone, välja arvatud väärtuse kustutamine ja liigutamine. Neid operatsioone võib sooritada vaid väärtuse omanik.

```
fn muuda(a: &mut String) {
      a.push_str("lisa"); // Lisame sõne lõppu uue sõne
}
```

Joonis 5. Näide sõnet muudetavalt laenavast funktsioonist.

Et vältida väärtuse samaaegset muutmist mitme laenaja poolt, võib väärtust muudetavalt laenata mistahes ajahetkel ülimalt üks kord [5].

1.4 Asünkroone käitus Rustis

Rusti ametlik blogi kirjutab, et alates stabiilse[1] Rusti versioonist 1.39.0 on toetatud ka asünkroone käitus. Seda võimaldavad `async` ja `await` võtmesõnad ning `Future` tagastustüüp [8]. Asünkroone käitus võimaldab programmil andmeid oodates tegeleda muude ülesannetega, näiteks andmebaasist vastust oodates kasutaja sisenditele reageerida.

Rusti asünkroonse programmeerimise õpik kirjutab, et asünkroone käitus on Rustis lisa-hinnata abstraktsioon (ingl *zero-cost abstraction*), mis tähendab, et asünkroonsete funktsioonide kasutamine ei vaja lisaressurssi võrreldes tavaliste funktsioonidega. Lisaks on asünkroonsed funktsioonid Rustis „laisad", ehk nad teevad tööd vaid siis, kui nende olekut päritakse (ingl *poll*) [9].

Kuigi `async` funktsioonid on Rustis sisseehitatud, ei saa neid kasutada ilma kolmanda osapoole poolt loodud tarkvara kastita[2] (ingl *crate*). Kõige populaarsem sellelaadne lahendus on tokio[3], mis implementeerib asünkroonsete funktsioonide kasutamiseks vajamineva loogika [9]. Põhjus, miks Rust ei sisalda ühte kindlat asünkroonse käituse implementatsiooni tuleneb vajadusest olla võimalikult paindlik. Kasutaja saab valida täpselt oma vajadustele vastava lahenduse või selle ka vajadusel ise implementeerida, hoides kokku väärtusliku ressurssi ning arendusaega.

Toetatud on ka klassikaline paralleelne käitus, kus programm jookseb samaaegselt mitmel operatsioonisüsteemi poolt antud lõimel. Üks peamisi tokio populaarsuse põhjusi peitubki asünkroonse ja paralleelse käituse sidumises, mis kombineerituna kompileeritavusest tuleneva jõudlusega on muutnud Rusti üheks kiiremini kasvavaks keeleks veebiserverite maailmas [10].

[1] Stabiilne Rust on Rusti versioon, kus kehtivad semantilise versiooninumbri poolt antavad garantiid.

[2] Tarkvara teek

[3] https://crates.io/crates/tokio

1.5 Ebaturvaline (ingl *unsafe*) Rust

Rusti veebiõpik defineerib ebaturvalise Rusti kui programmikoodi, mille kompileerimisel ei teosta kompilaator teatud kontrolle. See annab ebaturvalistele programmidele „supervõimed", mida tavalistel Rusti programmidel ei ole:

- Võimaluse otsendada (ingl *dereference*) ilma tüübita viitasid;
- Kutsuda välja teisi ebaturvalisi funktsioone;
- Ligipääsu staatilise eluajaga muutujatele;
- Implementeerida ebaturvalisi omadusi (ingl *trait*);
- Ligipääsu union andmestruktuuri väljadele.

Ebaturvalise koodijupi märkimiseks tuleb see asetada `unsafe` ploki sisse [7].

Selliste võimaluste vajadus tuleneb peamiselt asjaolust, et kompilaatori poolt tehtav staatiline analüüs on iseloomult konservatiivne. See tähendab, et lükatakse tagasi kõik ebakorrektsed programmid, isegi kui seetõttu ei kompileerita ka mõnda korrektset programmi [7]. Ebaturvaline Rust on viis ilmutatult kompilaatorile öelda, et antud koodijupi korrektsuse eest vastutab programmeerija ise. On rangelt soovituslik vältida `unsafe` kasutust nii palju kui võimalik, kuna iga kasutus lisab potentsiaalse vigade allika.

Oluline on mõista, et `unsafe` võtmesõna kasutamine annab vaid 5 eelnevalt kirjeldatud „supervõimet", kõik ülejäänud kontrollid teostab kompilaator ka ebaturvalise programmikoodi jaoks [7].

Ebaturvaline Rust leiab laialdaselt kasutust näiteks madala taseme süsteemiprogrammeerimisel. Võimalus otsendada tüübita viitasid lubab programmeerijal manipuleerida mälu ilma laenuhalduri poolt seatud piiranguteta ning töötada teistes keeltes kirjutatud koodiga. Joonisel 6 on toodud näide Rusti veebiõpikust, kus C keeles kirjutatud `abs()` funktsiooni kasutatakse Rusti koodis.

```
extern "C" {fn abs(input: i32) -> i32;}
fn main() {
    unsafe {println!("Absolute value according to C: {}",
abs(-3));}
}
```

Joonis 6. Näide C programmeerimiskeeles kirjutatud funktsiooni kasutamisest [7].

Välise funktsiooni väljakutse peab olema kindlasti **unsafe** ploki sees, kuna teised keeled ei järgi Rusti mälureegleid ja on seega Rusti vaatepunktist ebaturvalised [7].

1.6 Rusti tööriistad

Lisaks turvalisele mälumudelile on Rusti tugevuseks veel ka mugav arendajakogemus, mida võimaldavad mitmed standardiseeritud tööriistad.

Rusti installeerimine käib läbi tööriista nimega rustup[4], mis installeerib kõik arenduseks vajaliku. Olulisemad osad on näiteks kompilaator rustc, paketihaldur cargo ja keeleserver rust-analyzer. Lisaks võimaldab antud tööriist lihtsasti installeerida erinevaid kompilaatori versioone käsuga `rustup target add <platvorm>`, et võimaldada kompileerimist ka teistele platvormidele.

Peale installeerimist toimub enamik tööst läbi cargo, mis on ühtlasi nii Rusti paketihaldur kui ka ehitussüsteem (ingl *buildsystem*). Uue projekti saab initsialiseerida käsuga `cargo new`, mis loob uue kausta projekti algse struktuuriga. Automaatselt initsialiseeritakse git[5] repositoorium, luuakse *Cargo.toml* fail seadistamiseks ning *src* kaust *main.rs* failiga. Projekti kompileerimiseks piisab käsust `cargo build` ja programmi jooksutamiseks käsust `cargo run`, seda seni kuni aktiivne kaust sisaldab *Cargo.toml* faili. Kuna kõik Rusti projektid kasutavad ehitussüsteemina cargot, on äärmiselt lihtne teisi projekte nii jooksutada kui ka nendes navigeerida.

[4] https://rustup.rs/

[5] Versioonikontrolli süsteem

Läbi cargo uue projekti sõltuvuse lisamiseks saab kasutada käsku `cargo add <nimi>` või lisada vastav kirje manuaalselt Cargo.toml faili. Kompileerimisel laeb cargo automaatselt alla kõik vajaminevad paketid kesksest repositooriumist crates.io. Läbi cargo saab ka enda pakette repositooriumisse lisada, piisab vaid Cargo.toml korrektselt ülesseada, luua kasutaja crates.io keskkonda ning jooksutada käsk `cargo publish`.

Clippy on Rusti koodilintija[6], mis lisaks süntaktilistele vigadele hoitab kasutajat ka stilistiliste vigade eest. Clippy saab mugavalt installeerida läbi rustupi käsuga `rustup component add clippy`. Eriti soovituslik on clippy kasutamine just teistest keeltest tulevatele Rusti õppijatele, kuna see aitab tähelepanu juhtida koodijuppidele, mida saab Rusti konstruktsioonide abil efektiivsemalt kirjutada.

[6] Automaatne programmi- ja stiilivigade otsing lähtekoodist https://akit.cyber.ee/term/14603-linting

2. Logimisraamistiku loomine

Käesoleva töö praktilise osa esimene pool on logpeek nimelise logmisraamistiku loomine. Selle osa eesmärgiks on luua täisväärtuslik tarkvarateek, mis võimaldaks kasutajal võimalikult väikese vaevaga logida oma Rusti projektis toimuvat ja ühilduks Rusti ametliku logimise fassaadiga[7]. Viimasena avaldatakse valminud teek ka Rusti teekide repositooriumis.

2.1 Olemasolevate lahenduste puudused

Sarnaselt teiste programmeerimiskeeltega on Rusti jaoks juba varasemalt loodud mitmeid logimislahendusi, 24.03.2024 seisuga annab crates.io keskonnas võtmesõna „logging" otsimine 488 tulemust. Enamus neist on aga mõeldud suurtele projektidele ja nõuavad seega ka tunduvalt rohkem seadistust. Populaarseimad sellelaadsed näited on tracing[8] ja env_logger[9].

Tracing pakett võimaldab kasutajal jälgida pea kõike programmis toimuvat. Lisaks tavalisele logimisele on toetatud ka programmi sündmuste täpsem ajaline jälgimine (ingl *tracing*), mis võimaldab kiiresti diagnoosida programmis tekkinud vigu. Nõrkusteks on aga liigne keerukus ülesseadmisel, näiteks peab kasutaja täpse jälgimise võimaldamiseks järgima rangeid reegleid ning kasutama tracing poolt loodud konstruktsioone, ja võrdlemisi suur ressursikasutus.

Env_logger on ülesehituselt tunduvalt lihtsam, aga toetab konfigureerimist vaid läbi keskkonna muutujate (ingl *environment variables*), mis võib kohati osutuda üsnagi tülikaks.

Seevastu eksisteerib ka lahendusi mis seda probleemi vähemalt näiliselt lahendavad, populaarseim nendest on simplelog[10]. See teek sisaldab mitut erinevat logijat, näiteks faili või konsooli kirjutavat ning nende konfigureerimiseks kasutatavat objekti. Puudusteks on aga kehv konfiguratsiooni disain, kus iga logija jaoks tuleb luua uus konfiguratsiooni objekt, osad olulised seaded on liialt peidetud, logifaili haldamine on kasutaja vastutada ja toetatud pole

[7] Standardiseeritud viis Rusti rakenduste logimiseks. https://crates.io/crates/log

[8] https://crates.io/crates/tracing

[9] https://crates.io/crates/env_logger

[10] https://crates.io/crates/simplelog

logifaili lahkulöömine teatud suuruseni jõudmisel. Siiski on tegu hea lahendusega, mida antud praktilise osa loomisel jälgida ja parandada.

2.2 Tehniline teostus

See peatükk annab ülevaate logpeek nimelise logimisraamistiku loomise protsessist ja põhjendab arenduse käigus tehtud valikuid.

2.2.1 Projekti struktuur

Esmalt loodi projekti algne struktuur käsuga `cargo new --lib <projekti nimi>`. Lipp `--lib` ütleb cargole, et tegu on teek (ingl *library*) tüüpi projektiga ning *main.rs* asemel tuleks luua hoopis *lib.rs*. Siiski on testimiseks kasulik manuaalselt luua ka *main.rs* fail, cargo jätab selle automaatselt kõrvale.

Edasi jaotati projekt veel väiksemateks osadeks. Üldise konventsiooni kohaselt tohib `lib.rs` sisaldada vaid lõppkasutaja jaoks avalikku koodi, ehk sinna asetati tutvustav dokumentatsioon, logija initsialiseerimiseks kasutatava avalik funktsioon `init` ja konfiguratsiooni mooduli avalik eksport[11]. Lisaks loodi veel failid `config.rs` ja `logger.rs`, mis sisaldavad vastavalt konfiguratsiooniga ja logija endaga seotud koodi.

Erinevaid faile käsiteb Rust eraldi moodulitena, mis võimaldab koodi jaotada erinevatesse nimeruumidesse ja seeläbi kontrollida nendele ligipääsu. Kuna sooviti kasutajale kättesaadavaks teha konfigureerimiseks vajalikud konstruktsioonid, lisati `lib.rs` faili rida `pub mod config`. Logija implementatsioon sooviti aga hoida privaatsena, milleks lisati rida `mod logger`. Moodulites olevatele funktsioonidele ja struktuuridele saab ligi nimeruumi operaatoriga `::` või `use` võtmesõnaga.

2.2.2 Logija konfiguratsioon

Logija kasutajapoolne konfigureerimine käib läbi `Config` andmestruktuuri (ingl *struct*), mis koosneb eri valikuid kirjeldavatest väärtustikest (ingl *enum*). Väärtustikud toimivad Rustis veidi erinevalt kui teistes programmeerimiskeeltes, kuna iga väärtustiku element võib veel omakorda sisaldada mingit muud objekti. Joonisel 7 on toodud näide logide sihtkausta

[11] Teeb logija konfigureerimiseks kasutatava objekti kasutajatele kättesaadavaks.

seadistamiseks kasutatavast väärtustikust. `CurrentDir` puhul kirjutab logija logid aktiivsesse töökausta ja `Custom` puhul kasutaja poolt valitud kausta. Valik `Custom` on loogiliselt seotud failiteega, mis salvestatakse otse väärtuse sisse.

```
pub enum OutputDirName {
    CurrentDir,
    Custom(String),
}
```

Joonis 7. Sihtkausta seadistamiseks kasutatav väärtustik.

Sooviti ka, et kasutaja saaks mugavalt luua uue `Config` objekti vaikeseadetega, et veelgi vähendada seadistamisele kuluvat vaeva. Selleks implementeeriti `Config` andmestruktuurile `Default` nimeline omadus (ingl *trait*). Rusti omadused on oma põhimõttelt sarnased Java liidestele, kus omaduse implementeerimiseks tuleb luua teatud meetodid. Antud juhul on selleks meetodiks `default`, mis tagastab `Config` tüüpi objekti eelnevalt valitud vaikeseadetega.

2.2.3 Logija implementeerimine

Et logija oleks kasutatav Rusti logimise fassaadi poolt peab ta implementeerima `Log` omadust, mis on defineeritud log teegis. Selleks tuleb luua 3 meetodit:

- `enabled`, mis otsustab sõnumi taseme[12] põhjal, kas antud sõnum logitakse või mitte;
- `log`, mis edastab sõnumi edasi meie logijale;
- `flush`, mis võimaldab kasutajal vajadusel manuaalselt logid kettale kirjutada.

Esimene ja viimane neist on triviaalsed ja ei erine teistest logijatest. Meetod `log` kontrollib esmalt, kas antud logikirje tuleks logijale edastada kasutades eelmainitud `enabled` meetodit ja seejärel formaadib ning edastab sõnumi logijale.

Logija ise on defineeritud kui andmestruktuur koos vastavate meetoditega. Andmestruktuur `Logger` sisaldab endas välju `output_lock`, `config` logija konfiguratsiooni objektiga ning `custom_time_format`, mis leiab kasutust juhul kui kasutaja soovib defineerida oma enda ajatempli formaadi.

[12] Võimalikud tasemed on ERROR, WARN, INFO, DEBUG ja TRACE järjestatud kõrgeimast väiksemani.

Vaatame lähemalt `output_lock` välja, mille andmetüübiks on `Mutex<Option<Output>>`. Tegu on muteksiga, mis kaitseb logide kettale kirjutamiseks kasutatavat objekti andmejooksude eest mitmelõimelistes programmides. Nende vahele on lisatud `Option`[13] väärtustik olukorra jaoks, kus kasutaja soovib logisi kirjutada vaid standard väljundvoogu[14], millisel juhul `Output` objekti ei initsialiseeritagi.

Logija objekti saab luua staatilise meetodiga `Logger::new(config: Config)` andes sellele sisendiks konfiguratsiooni objekti. Oluline on, et antud meetod ei laena `Config` objekti, vaid liigutab seda. See tagab, et konfiguratsiooni ei saa peale objekti loomist enam ootamatult muuta.

Logide kirjutamisega tegeleb meetod `write`. Esmalt kontrollib antud meetod, kas kasutaja soovib logisi kirjutada konsooli ja/või faili vaadates logija loomisel kasutatud `Config` objekti. Seejärel lisatakse sõnumile sõltuvalt kasutaja soovist ANSI paojadad[15] (ingl *escape sequence*) logi tasemete värvi järgi eristamiseks ja kirjutatakse sõnum standard väljundvoogu.

Faili kirjutamiseks tuleb esmalt pärida muteksi lukk. Joonisel 8 on toodud lihtsustatud näide luku pärimisest. Luku pärimine käib läbi muteks objekti `lock` meetodi, mis vajadusel blokeerib antud lõime töö seniks, kui teine lõim luku vabastab. Seejärel tagastatakse viide muteksi poolt kaitstavale sisemisele väärtusele. Lukk vabastatakse automaatselt skoobi lõpus, vältimaks tupikut[16].

```
if let Some(sisemine_väärtus) = muteks.lock().unwrap() {
    // sisemine_väärtus on kasutatav vaid siin skoobis
}
```

Joonis 8. Näide muteksi luku pärimisest.

[13] Omab kahte võimaliku väärtust: None ja Some(sisemine_väärtus).
[14] Andmevoog, kuhu kirjutatakse vaikimisi programmi poolt väljastatav informatsioon.
[15] Sümbolite jadad, mis omavad erilist tähendust. Näiteks ESC[?25l teeb kursori nähtamatuks.
[16] Olukord, kus üks protsessi lõim hoiab muteksi lukku lõpmatult.

Koodi kvaliteedi tagamiseks lisati logija juurde ka erinevad testid. Rustis on tavaks kirjutada kõik testid samadesse failidesse, kus asuvad testitavad objektid ja funktsioonid. Selleks tuli deklareerida uus moodul `tests` koos päisega `#[cfg(test)]`. Antud päis annab kompilaatorile märku, et tegu on testidega ja järgnevat koodi ei pea tavaolukorras kompileerima. Testide jooksutamiseks saab kasutada käsku `cargo run test`, mis käivitab ükshaaval `test` päisega funktsioone ja väljastab tulemuse vastavalt sellele, kas antud funktsioon lõpetas töö edukalt või viskas erindi.

2.2.4 Avaldamine crates.io keskkonnas

Arendusprotsessi viimase sammuna avalikustati valminud teek ka Rusti teegi repositooriumis. Selleks tuli `Cargo.toml` faili täiendada nõutud väljadega, näiteks autorite loetelu, paketi kirjeldus ja lähtekoodi VCS (*Version Control System*) repositooriumi link.

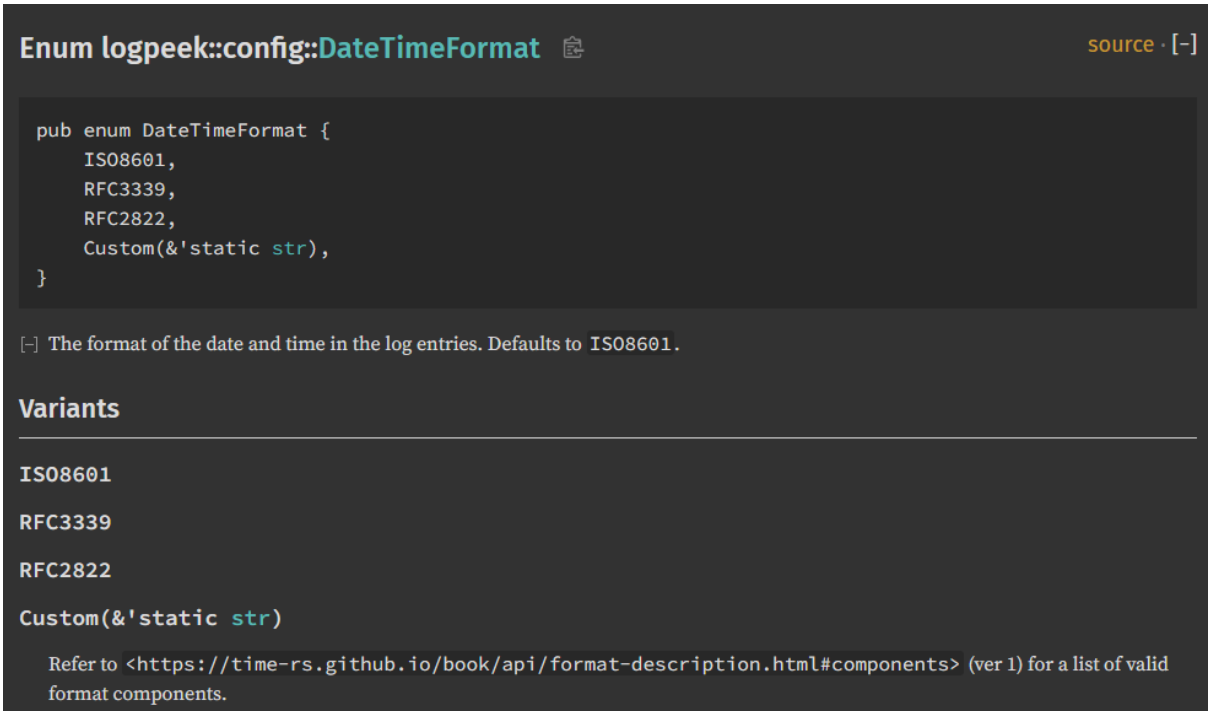
Järgnevalt tuli luua kasutaja crates.io keskkonda, mis töö kirjutamise hetkel käis läbi Githubi[17] ühenduse, ning luua omale API (*Application Programming Interface*) tooken. Loodud tookeni sai mugavalt salvestada käsuga `cargo login`. Viimase sammuna tuli jooksutada käsku `cargo publish`, mille tulemusel pakiti projekt `.crate` formaati ning laeti automaatselt üles repositooriumisse.

Peale üles laadimist genereeritakse automaatselt ka projekti dokumentatsioon ning lisatakse see docs.rs keskkonda. Soovi korral võib mistahes cargo projekti dokumentatsiooni ka lokaalselt genereerida, kasutades käsku `cargo doc` valikulise lipuga `--open`, mis avab valminud veebilehe brauseris. Tulemuseks on ülevaade projekti avalikest funktsioonidest ja struktuuridest, millele saab selgitavat infot lisada läbi `///` kommentaaride otse lähtekoodis. Joonisel 9 on toodud ajaformaadi konfigureerimise väärtustiku lähtekood ning joonisel 10 sellele vastav genereeritud dokumentatsioon.

[17] https://github.com

```rust
/// The format of the date and time in the log entries.
/// Defaults to `ISO8601`.
pub enum DateTimeFormat {
    ISO8601,
    RFC3339,
    RFC2822,
    /// Refer to
`<https://time-rs.github.io/book/api/format-description.html#compo
nents>` (ver 1) for a list of valid format components.
    Custom(&'static str),
}
```

Joonis 9. Ajatempli formaadi konfigureerimiseks kasutatav väärtustik.



Joonis 10. Kuvatõmmis cargo poolt genereeritud dokumentatsioonist.

Selline lähenemine on loonud olukorra, kus pea kõik Rusti projektid kasutavad samasugust dokumentatsiooni formaati, võimaldades kasutajatel vähese vaevaga nendes navigeerida.

2.2.5 Tulemused ja loodud raamistiku kasutamine Rusti projektides

Valminud logimisraamistik võimaldab kasutajal vabalt seadistada järgnevat:

1. Loodava logifaili nime ja sihtkausta;
2. Logitavate sõnumite miinimum taset;
3. Ajatsooni;
4. Sõnumite logimist väljundvoogu ja/või faili;
5. Sõnumite logimist vea- ja/või väljundvoogu;
6. Ajatempli formaati;
7. ANSI paojadade kasutust sõnumite väljundvoogu kirjutamisel;
8. Sünkroonset või asünkroonset logide faili kirjutamist;
9. Uue logifaili loomist kindla faili suuruseni jõudmisel;
10. Logiallikate välja filtreerimist.

Lisaks parandab valminud lahendus ka simplelog teegi puudusi, hallates logifaili täielikult automaatselt, kasutades vaid ühte konfiguratsiooni objekti kogu logija seadistamiseks ning võimaldades kasutajal seadistada soovitud maksimaalset logifaili suurust.

Logimisraamistiku kasutamiseks tuleb esmalt lisada log ja logpeek teegid projekti sõltuvustena, mida saab mugavalt teha käsuga `cargo add log logpeek`. Soovi korral võib teegi ka ise kompileerida, lähtekood on saadaval lisas I olevas repositooriumis.

Seejärel saab logija initsialiseerida `logpeek::init()` funktsiooniga. Et muuta seadistuse vaikeväärtusi, tuleb esmalt luua `Config` objekt soovitud väljadega ning seejärel see `init()` funktsioonile sisendiks anda. Sellega on seadistus tehtud ning logija on kasutatav rakenduse üleselt läbi log teegis defineeritud funktsioonide.

2.3 Potentsiaalsed edasiarendused

Eesmärgiks oli luua võimalikult lihtne ja mugav logimise lahendus, mida valminud lahendus ka on. Siiski võib tulevikus projektile lisada rohkem valikuid logija seadistuseks või implementeerida tracing paketiga sarnane funktsionaalsus vigade paremaks diagnoosimiseks.

Hetkel kasutab loodud lahendus Rusti standardteeki (ingl *standard library*), mis sisaldab näiteks failidega töötamise funktsioone ja mitmelõimeliste programmide jaoks vajaminevat `Mutex` andmestruktuuri. Kuna standardteek toetub suuresti operatsioonisüsteemile, siis ei saa seda kasutada manussüsteemide tarkvara arendamisel.

Et muuta loodud raamistik kasutatavaks ka standardteegi puudumisel, tuleb sellest kompilaatorile märku anda `#![no_std]` päise lisamisega. Sellisel juhul asendatakse standardteek operatsioonisüsteemist sõltumatu core[18] teegiga. See aga tähendaks, et mitmed primitiivid tuleks ise nullist implementeerida ja kogu projekt suuresti ümber mõelda.

Rusti teekidele on võimalik lisada erinevaid lippe (ingl *feature flags*)[19], mis võimaldavad kasutajal täpselt valida, millist võimekust ta antud paketilt soovib. Üldiselt pannakse selliste lippude taha ressursi nõudlikumad lisad, näiteks kolmandatest teekidest sõltuv funktsionaalsus. Üheks selliseks lipuks on konventsiooni kohaselt `no-std`, mis tähendab võimekust töötada ilma standardteegita. Just `no-std` lipu võiks ka valminud paketile lisada ning luua erinevad lipud valikuliste sõltuvuste jaoks.

[18] https://doc.rust-lang.org/core
[19] https://doc.rust-lang.org/cargo/reference/features.html

3. Veebirakenduse loomine

Käesoleva töö teine praktiline osa on logpeek-server nimelise veebirakenduse loomine, mis võimaldaks kasutajal mugavalt hallata erinevate rakenduste logifaile. Valminud lahendus peab olema kasutatav nii lokaalselt kui ka üle võrgu, toetama võimalikult palju erinevaid logide formaate, töötama nii Windowsi, Linuxi kui ka MacOSi peal ning vajama võimalikult vähe käitusaegset ressurssi.

3.1 Olemasolevate lahenduste puudused

Populaarseimad logide kogumise ja filtreerimise lahendused on mitmest erinevast osast koosnevad ELK[20] ning Grafana, Loki ja Promtaili kombinatsioon. Kuigi ettevõtetele ja suurte andmehulkadega klientidele pakuvad antud lahendused pea kõik vajaliku, pole väikekasutajatele need lahendused mõistlikud oma keerukuse ja ressursi nõudlikuse tõttu.

Tavakasutajale suunatud lahendusi eksisteerib tunduvalt vähem ning enamus neist on kas tasulised või puuduva funktsionaalsusega. Antud praktilise osa raames valmiv tarkvara loodab neid vigu parandada, olles avatud lähtekoodiga ja pakkudes lisa funktsionaalsust võrreldes logifailide tekstiredaktoris vaatamisega.

3.1 Kasutatud tarkvara

Järgnevalt antakse ülevaade veebirakenduse loomisel kasutatud tarkvarast.

3.1.1 Axum

Axum[21] on tokio meeskonna poolt loodud Rusti veebiraamistik, mis on disainitud olema võimalikult modulaarne ning põhineb täielikult tokio organisatsiooni poolt loodud tarkvaral [11]. Antud projekti jaoks sai Axum valitud suurepärase asünkroonse käituse toe, hea dokumentatsiooni ja teiste arendajate soovituste tõttu. Lisaks tähendab antud raamistiku populaarsus seda, et lahendused tavalisematele probleemidele on internetist lihtsasti leitavad ja vajadusel saab kiiresti abi küsida erinevatest foorumitest.

[20] Elasticsearch, Logstash, ja Kibana.
[21] https://docs.rs/axum/latest/axum/

3.1.2 Vue.js

Vue[22] on laialdaselt kasutatud JavaScripti raamistik, mille põhilisteks tegevusteks võrreldes teiste raamistikega on kasutajamugavus, suurepärane jõudlus ja paindlikkus. Antud raamistiku valikut soodustas veel autori varasem kokkupuude ning töökogemus, mis võimaldasid arenduse põhirõhu panna just Rusti, mitte JavaScripti peale.

3.2 Tehniline teostus

See peatükk annab ülevaate logpeek-server nimelise veebirakenduse loomise protsessist ja rakenduse arhitektuurist.

3.2.1 Veebirakenduse arhitektuur

Projekt jaguneb Rustis kirjutatud tagakomponendiks (ingl *back-end*) ja Vuega loodud kliendipoolseks komponendiks (ingl *front-end*), mis suhtlevad omavahel HTTP (*Hypertext Transfer Protocol*) päringute abil.

Tagakomponendi ülesanneteks on logifailides oleva info mällu lugemine, logikirjete parsimine, filtreerimine ja kliendipoolsest komponendist tehtavatele päringutele vastamine. Tegemist on ,,laisa" serveriga, mis teeb tööd vaid siis kui kasutaja sellele päringuid saadab. Selline disaini otsus tulenes asjaolust, et failide kettalt lugemine on reeglina aeglane operatsioon ning võib sõltuvalt logide arvust väiksema võimsusega süsteemidele mõjuda halvavalt. Muutes serveri ,,laisaks", teab kasutaja rakenduse avamisel sellega arvestada.

Kliendipoolse komponendi ülesanneteks on kasutajaliidese pakkumine tagakomponendiga suhtlemiseks ja info kasutajasõbralik esitamine. Peamisteks komponentideks on infopaneeli vaade, mis annab ülevaate viimase 24 tunni ja 7 päeva jooksul toimunust ning logitabel, mis esitab logikirjed tabeli vormis koos valikuliste filtritega.

Lõpptulemus on üksik binaarfail, mis sisaldab endas nii taga- kui ka kliendipoolset komponenti. See säästab kasutajat keerukast seadistusest, lihtsustab märgatavalt projekti kasutajatele levitamist ja võimaldab installeerimise vaba kasutust.

[22] https://vuejs.org/

3.2.2 Tagakomponendi arhitektuur

Tagakomponent ehk server on loodud Rustis kasutades Axum raamistikku. Üldine ülesehitus järgib REST (*Representational State Transfer*) filosoofiat, kus serverist info pärimiseks on loodud vastavad teed (ingl *endpoint*), millele kliendipoolne komponent päringuid saadab. Lisaks on serveril veel mitu vahetarkvara (ingl *middleware*) kihti, mis töötlevad igat sissetulevat päringut ennem nende ruuterile[23] (ingl *router*) edastamist. Järgnevalt kirjeldatakse kolme olulisimat teed ja autentimise vahevara kihti eraldi.

Esimene tee, millele kasutaja päringu teeb, on `index_handler`, mis tagastab kliendipoolse komponendi kuvamiseks vajalikud failid. Täpsemalt tagastab see tee vaid *index.html* faili, mille põhjal teeb kasutaja veebilehitseja edasised päringud vajalike failide jaoks `static_handler` teele. Kõik vajalikud kliendipoolse komponendi failid on rust-embed[24] nimelise teegi abiga kompileeritud otse binaarfaili.

Tee `dashboard_info`, mille ülesandeks on mälus olevate logikirjete põhjal välja arvutada kasutajale infopaneeli vaatel kuvatav info. Peale info välja arvutamist ja vastavat töötlemist edastatakse see kliendipoolsele komponendile joonisel 11 toodud andmestruktuuriga. Siin on olulisel kohal `#[derive(Serialize)]`, mis tuletab[25] antud andmestruktuurile automaatselt serde[26] teegis defineeritud `Serialize` omaduse ja võimaldab Axumil kasutada defineeritud andmestruktuuri päringule vastamiseks JSON (*JavaScript Object Notation*) formaadis.

[23] Tarkvaras implementeeritud konstruktsioon, mille ülesanne on sissetulevad päringud suunata vastavatele funktsioonidele. Veebiarenduses laialdaselt kasutatud.
[24] https://crates.io/crates/rust-embed
[25] Lihtsamate omaduste automaatne implementeerimine andmestruktuurile.
[26] Andmete serialiseerimise ja deserialiseerimise teek. https://serde.rs/

```rust
#[derive(Serialize)]
pub struct DashboardResponse {
    total_logs_24: [u32; 24],
    error_logs_24: [u32; 24],
    warning_logs_24: [u32; 24],
    total_logs_week: [u32; 7],
    error_logs_week: [u32; 7],
    warning_logs_week: [u32; 7],
    top_modules_24: Vec<(String, f32)>,
    top_modules_week: Vec<(String, f32)>,
    log_buffer_usage: f32,
    total_log_entries: u32,
}
```

Joonis 11. Infopaneeli päringu vastust kirjeldav andmestruktuur.

Järgmiseks vaatame `log_table` teed, mille ülesandeks vastavalt kasutaja poolt seatud filtritele mälus olevate logikirjete leidmine. Töö optimeerimiseks on kasutusel logikirjete lehendamine (ingl *pagination*), mis võimaldab kasutajale kuvada vaid teatud arvu logikirjeid korraga. See vähendab märgatavalt iga päringuga tehtavat tööd, aga suurendab päringute arvu. Kliendipoolsele komponendile tagastatakse järjend vastavate logikirjetega JSON formaadis.

Autentimise vahevara kihi ülesandeks on otsustada, kas suunata antud päring edasi järgmisesse vahevara kihti või lükata päring tagasi vastava veakoodiga. Autentimine põhineb HTTP Auth[27] päisel (ingl *header*), kus igale päringule lisatakse Base64[28] formaadis kasutajanime ja parooli kombinatsioon. Kuigi avaliku veebi jaoks on selline autentimine selgelt ebaturvaline, siis lokaalse võrgu jaoks piisav.

3.2.3 Logikirjete töötlemine

Kogu rakenduse tuumaks on logifailide kettalt lugemine, nende parsimine ning mällu salvestamine. Iga jälgitava rakenduse jaoks on mälus eraldi ringpuhver[29], mis sisaldab antud rakenduse logikirjeid parsitud andmestruktuuridena. Iga logi kirje jaoks salvestatakse

[27] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization
[28] Teksti kodeering.
[29] Esimesena sisse, esimesena välja andmestruktuur mille täitumisel kirjutatakse vanimad kirjed üle.

ajatempel, tase, allikas, sõnum ja jälgitava rakenduse indeks. Indeks vastab jälgitava rakenduse nimele paisketabelis, mis võimaldab logikirje salvestamisel mälu kokku hoida ning nime vajadusel sõnekujule tagasi tõlkida. Joonisel 12 on toodud ühte logikirjet väljendav andmestruktuur.

```
struct LogEntry {
    #[serde(with = "time::serde::rfc3339")]
    timestamp: OffsetDateTime,
    level: log::Level,
    module: String,
    message: String,
    application: usize,
}
```

Joonis 12. Logikirjet väljendav andmestruktuur.

Kasutaja peab iga jälgitava rakenduse jaoks sätestama kindla puhvri suuruse, mis vastab korraga mälus hoitavate logikirjete arvule. Veebirakenduse esmasel käivitamisel tuleb vastavad puhvrid luua ning need paigutada jagatud paisketabelisse. Puhvrite loomisel allokeeritakse koheselt vajaminev mälu täidetud puhvrite salvestamiseks. Selline lähenemine võimaldab kasutajale võimalikult varakult teada anda, kui olemasolevast mälust ei piisa ja puhvrite suurust tuleks vähendada.

Logifailide esmase lugemise optimeerimiseks töödeldakse logifaile kaks korda. Esmalt sorteeritakse kasutaja poolt sätestatud failiteel olevad failid viimase muutmise aja järgi, mille saab mugavalt pärida operatsioonisüsteemi käest. Järgnevalt hakatakse lugema nendes failides olevaid ridu alustades viimati muudetud failist, milleks tuleb fail täielikult läbi käia. Seda tehakse seni, kuni kõik failid on läbi vaadatud või ridade arv on puhvri mahutavusest suurem. See võimaldab jätta töötlemata logifailid, mis puhvrisse ei mahu.

Edasi on protsess sama nii logifailide algse lugemise kui ka käitusaegse lugemise jaoks. Sorteeritud logifailidest jäetakse kõrvale need, mille viimase muutmise aeg on sama selle faili eelmisel töötlemisel salvestatud ajaga. Lisaks viimase muutmise ajale salvestatakse iga faili

jaoks ära ka loetud read, vältides logikirjete topelt töötlemist ja võimaldades varasemalt loetud read vahele jätta.

Iga failist loetud rida parsitakse logikirjet kirjeldavaks andmestruktuuriks. Selleks kasutatakse kasutaja poolt sätestatud regulaaravaldist. Regulaaravaldis on jaotatud nimega gruppideks, mis võimaldab vajadusel nende järjekorda muuta ning teeb avaldise tunduvalt loetavamaks. Joonisel 13 on toodud näide võimalikust logikirjest ning sellele vastavast regulaaravaldisest.

2024-01-17T14:12:25+02:00 INFO logpeek_server - Listening on http://0.0.0.0:3001

^(?P<timestamp>\S+) (?P<level>\S+) (?P<module>\S+) - (?P<message>.+)$

Joonis 13. Logikirje ja sellele vastav regulaaravaldis.

Ainus regulaaravaldise grupp, mis peab kindlasti olema sätestatud on <message>, ehk lihtsaim võimalik regulaaravaldis mis vastab mistahes logikirjele on ^(?P<message>.*)$. Sellisel juhul antakse puuduvatele gruppidele vaikeväärtused, ajatempel on logikirje töötlemise aeg, tase on alati INFO ja allikaks „N/A”.

Kogu logikirjete töötlemise protsess toimub nii rakenduse esmasel käivitamisel kui ka iga serverisse tehtava päringuga. Liigse töö vältimiseks on kasutajal võimalik sätestada minimaalne aeg logide uuendamise vahel, vaikeväärtusena on see 10 sekundit. Vajadusel saab puhvrite uuendamise protsessi käivitada ka kasutajaliideses oleva nupuga, mis seda piirangut eirab.

3.2.4 Rakenduse konfigureerimine

Konfigureerimiseks on kasutusel config[30] teek, mis toetab konfiguratsiooni lugemist mitmest erinevast allikast. Olulisimaks neist on võimalus lugeda TOML[31] (*Tom's Obvious Minimal Language*) formaadis faile, mis on samuti ka cargo konfigureerimiseks kasutatav formaat ja seega juba kõigile Rusti arendajatele tuttav.

[30] https://docs.rs/config/0.14.0/config/index.html
[31] Lihtne, inimloetav konfiguratsiooni formaat.

TOML väärtused on jaotatud võtmete abil sektsioonideks. Joonisel 14 on toodud logimise seadistamiseks kasutatav sektsioon. Siin on `main.logger` sektsiooni võti ning järgnevad väljad vastavad väärtused. Et lugeda näiteks `log_path` välja väärtust, peame sellele viitama koos sektsiooni võtmega, ehk antud juhul `main.logger.log_path`. Võimalik on lisada ka kommentaare, muutes faili veelgi loetavamaks.

```
[main.logger]
# When true, sets the minimum log level to debug
enable_debug = false
# Whether to write log files
log_to_file = true
# When log_to_file is true, logs will be written to this directory
log_path = "logpeek-logs"
```

Joonis 14. Näide TOML faili sektsioonist.

Veebirakenduse kasutamiseks peab kasutaja seadistama jälgitavate rakenduste kohta käiva info. Seda saab teha `[[application]]` sektsioonide abil, mida võib olla piiramatu arv. Iga sektsiooni alla tuleb lisada kohustuslikud väljad logifailide asukohaga ja logikirjete parsimiseks kasutatava regulaaravaldisega. Veel saab kasutaja seadistada logikirje ajatempli formaadi ja soovitud puhvri suuruse.

Toetatud on ka keskkonna muutujatele põhinev konfiguratsioon, kuid selliselt saab seadistada vaid ühe jälgitava rakenduse, kuna kahte identse võtmega keskkonnamuutujat olla ei tohi. Eriti kasulik on keskkonna muutujatel põhinev seadistus ühe TOML seadistusfaili jagamisel mitme süsteemi vahel, kuna keskonnamuutujad on suurema prioriteediga kui failist loetud väljad, võimaldades süsteemi põhist seadistust ilma faili muutmata.

3.2.5 Kompileerimine erinevatele platvormidele

Kuigi rustup omab sisseehitatud võimalust installeerida süsteemiarhitektuurist erinevaid kompilaatori versioone ning neid mugavalt hallata käsuga `rustup target add <target>`, siis tihtipeale pole see piisav kogu projekti kompileerimiseks. Näiteks proovides projekti kompileerida Windowsi masinal x86_64-unknown-linux-gnu arhitektuuri jaoks, saame vea puuduva GCC[32] (*GNU Compiler Collection*) kompilaatori versiooni kohta.

[32] Kompilaator C ja C++ programmeerimiskeeltele.

Antud juhul on selle põhjustajaks teek nimega ring[33], mis võimaldab HTTPS (*Hyper Text Transfer Protocol Secure*) tuge. Täpsemalt soovib ring teek kompileerida oma C programmeerimiskeeles kirjutatud sõltuvusi, mis nõuavad süsteemi arhitektuurile vastavat kompilaatori versiooni.

Selle probleemi lahendamiseks saame kasutada cargo lisa nimega cross[34], mis võimaldab lihtsasti kompileerida programme teiste arhitektuuride jaoks. Selleks kasutab cross spetsiaalseid Docker[35] konteinereid, kuhu on installeeritud vaid sihtmärgiks oleva arhitektuuriga operatsioonisüsteem ning vastav Rusti kompilaatori versioon. Eelneva vea lahendamiseks peame kasutama `cargo build` asemel käsku `cross build`, mille tagajärjel laeb cross automaatselt alla õige konteineri, käivitab selle aktiivses töökaustas ning kompileerib projekti.

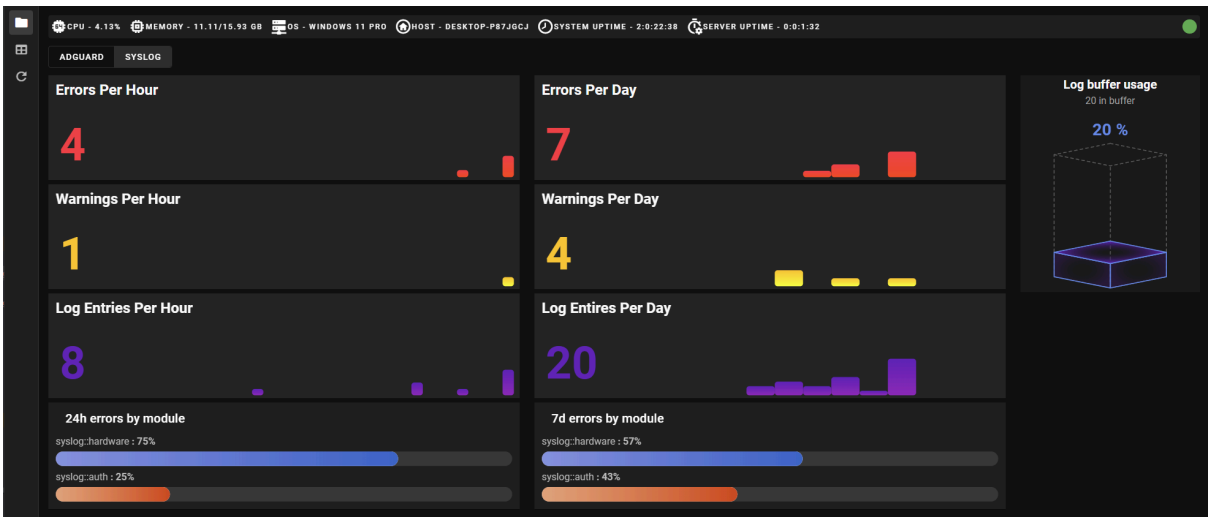
3.2.6 Kliendipoolse komponendi loomine

Kuna käesoleva töö põhifookuses on Rust programmeerimiskeel, siis antakse kliendipoolse komponendi loomise protsessist vaid lühiülevaade.

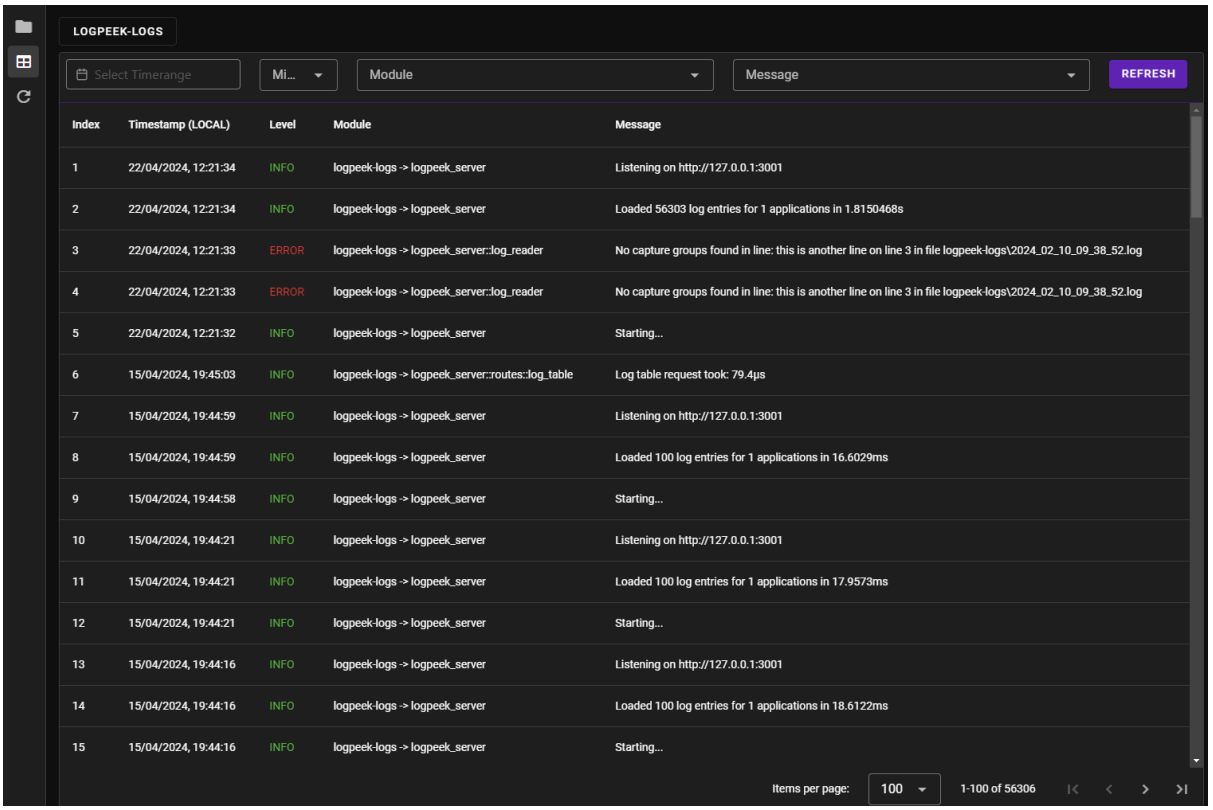
Olles esmalt installeerinud Node.js[36] tarkvara, saab NPM (*Node Package Manager*) abiga luua projekti algse struktuuri. Seda saab teha käsuga `npm create vite@latest`, mis seab automaatselt üles kõik vajaliku ning võimaldab koheselt arendusega alustada. Tasub ära märkida, et antud juhul kasutati Vite[37] projektimalli, mis on Vue autorite poolt loodud arendusserver ning ehitussüsteem. Vite eelistena teiste lahenduste ees tuuakse välja väikest käivitusaega, paremat ressursikasutust läbi projekti mooduliteks jagamise ja kiiret HMR[38] (*Hot Module Replacement*) tuge [12].

Edasi jaotati projekt kolmeks vaheleheks. Nendeks on avaleht koos süsteemiinfo ja näidikupaneeliga (vt. Joonis 15), logide filtreerimise ja sirvimise lehekülg (vt. Joonis 16) ning autentimiseks kasutatav sisselogimise leht.

[33] https://docs.rs/ring/latest/ring/
[34] https://github.com/cross-rs/cross
[35] Konteinerisatsiooni tarkvara. https://www.docker.com/
[36] https://nodejs.org/en
[37] https://vitejs.dev/
[38] Võimaldab kuvada projektis tehtavaid muudatusi ilma arendusserveri taaskäivituseta.



Joonis 15. Kliendipoolse komponendi avaleht.



Joonis 16. Logide filtreerimise ja sirvimise vaade.

Nagu eelnevalt mainitud, suhtleb kliendipoolne komponent serveriga läbi HTTP päringute. Kuna kasutusel on TypeScript[39], on tavaks nende päringute vastused selgelt defineerida läbi liideste (ingl *interface*), et vältida potentsiaalseid käitusaegseid vigu. Antud lähenemine

[39] Microsofti poolt loodud JavaScripti versioon, mis kasutab ilmutatud tüübisüsteemi.

muudab ka koodi loetavamaks, kuna iga fetch[40] väljakutse on selgelt annoteeritud oodatud tagastustüübiga.

Viimaseks sammuks on projekti ehitamine. Selleks tuleb kasutada käsku `npm run build`, mis kompileerib projekti Vite abiga JavaScripti ja CSS (*Cascading Syle Sheets*) failideks. Just need failid lisatakse tagakomponendi poolt loodavasse binaarfaili rust-embed teegi abil ja võimaldavad kogu rakendust levitada ühe, installeerimist mittevajava failina.

3.2.7 Valminud tarkvara funktsionaalsus ja kasutamine

Valminud tarkavara pakub kasutajatele järgnevat funktsioonalsust:

1. Kõikvõimalike logifailide haldust ühest kohast;
2. Logikirjete filtreerimist aja, taseme ja sisu järgi;
3. Ülevaadet logifailides toimuvast näidikupaneeli näol;
4. Võimalust vaadata süsteemi logisi tervikuna või iga rakenduse jaoks eraldi;
5. Valikulist autentimist rakendusele ligipääsuks;
6. Võimalust sätestada iga hallatava rakenduse logipuhvri suurust;

Lisaks on valminud veebirakendus loodud olema võimalikult väikese ressursi vajadusega, tehes tööd vaid siis, kui kasutaja seda soovib. Seetõttu sobib logpeek-server kasutamiseks ka platvormidel nagu RaspberryPi[41], kus saadaolevaid ressursse on vähe.

Veebirakenduse kasutamiseks tuleb esmalt hankida selle binaarfail. Selleks on võimalik kogu rakendus ise kompileerida või laadida alla varasemalt kompileeritud binaarfail GitHubist. Lähtekoodi repositooriumi aadress on leitav lisas I.

Järgnevalt tuleb veebirakendus korrektselt seadistada. Seadistusfaili malli loomiseks on lihtsaim viis rakendus korra käivitada, mille tulemusel luuakse aktiivsesse töökausta *config.toml* fail. Seejärel saab loodud faili avada tekstiredaktoris ning lisada hallatavate logifailide töötlemiseks vajalikud väljad, milleks on logifailide asukoht ning parsimiseks kasutatav regulaaravaldis.

[40] JavaScripti funktsioon, mis sooritab HTTP päringuid.
[41] Pisike, ühest trükiplaadist koosnev arvuti. https://www.raspberrypi.com/

Kui kasutaja soovib rakendust kasutada ka üle võrgu, tuleks lisada veel soovitud veebiaadress. Autentimise võimaldamiseks tuleb lisada seadistusfaili vastavasse väja salasõna, vastasel juhul on rakendusele ligipääs kõigil, kes on ühendatud sama võrguga.

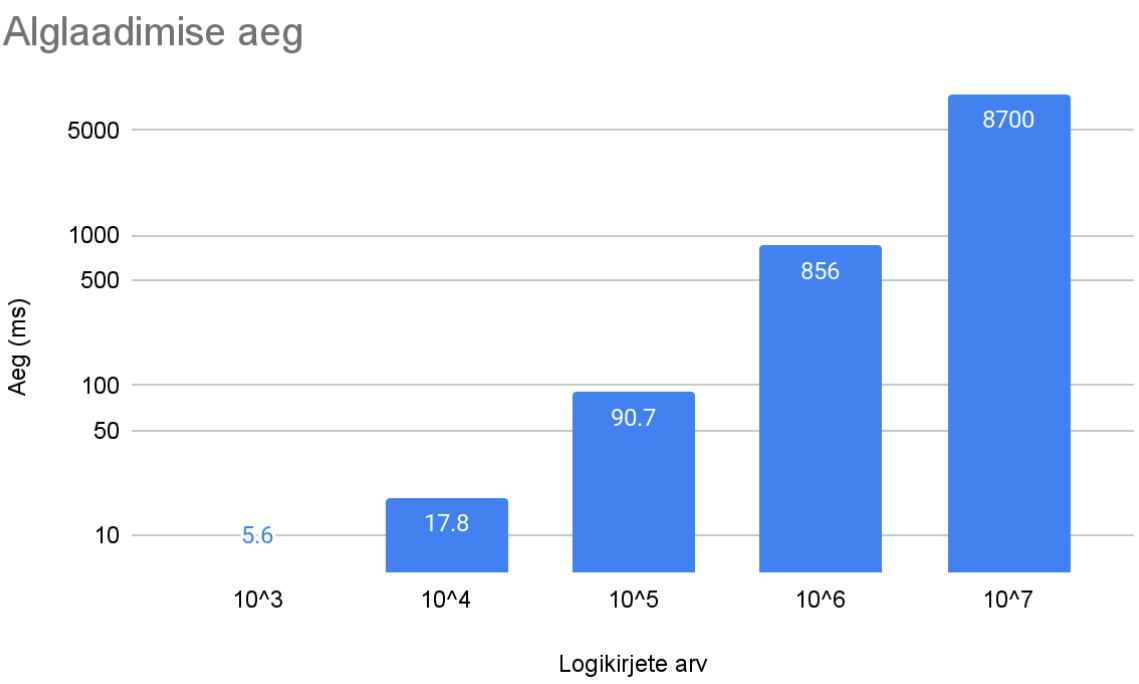
Rakenduse käivitamiseks tuleb binaarfaili jooksutada koos seadistusfaili asukohaga. Kui seadistusfail asub binaarfailiga samas kaustas ning on failinimega *config.toml*, siis leiab rakendus selle automaatselt üles. Kasutajaliidesele ligipääsuks tuleb sisestada terminali aknas kuvatav aadress veebilehitsejasse.

3.3 Jõudlustestimine

Valminud rakenduse kvaliteedi hindamiseks viidi läbi jõudlustestimine. Testimiseks kasutatavad logifailid genereeriti kasutades töö esimeses praktilises osas valminud logimisteeki. Iga logifaili rida sisaldab konstantset, 14 tähemärgist koosnevat sõnet koos varieeruva loendur numbriga. See annab hea üldistuse logifailides esinevast entroopiast, kus üldine info on suuresti sama ja read erinevad üksteisest vaid väikeste detailide poolest. Parsimiseks kasutatati täieliku regulaaravaldist, mis tähendab, et igast logikirjest eraldati kogu saadaolev informatsioon. See on parsimiseks kuluva aja suhtes halvim võimalik juht.

Testimiseks kasutati Windows 11 Pro arvutit Ryzen 5600 6-tuumalise protsessori, 16GB DDR4 mälu ja 1TB NVMe SSDga. Rakendust testiti kokku 5 erineva logifailiga, milles oli vastavalt 1 000, 10 000, 100 000, 1 000 000 ja 10 000 000 logikirjet. Testitavateks kategooriateks olid rakenduse alglaadimise ehk logipuhvrite esimeseks täitmiseks kulunud aeg, rakenduse tööaegne mälukasutus võrreldes logifaili suurusega kettal ning kasutaja päringutele vastamiseks kuluv aeg nii parima kui halvima juhu korral.

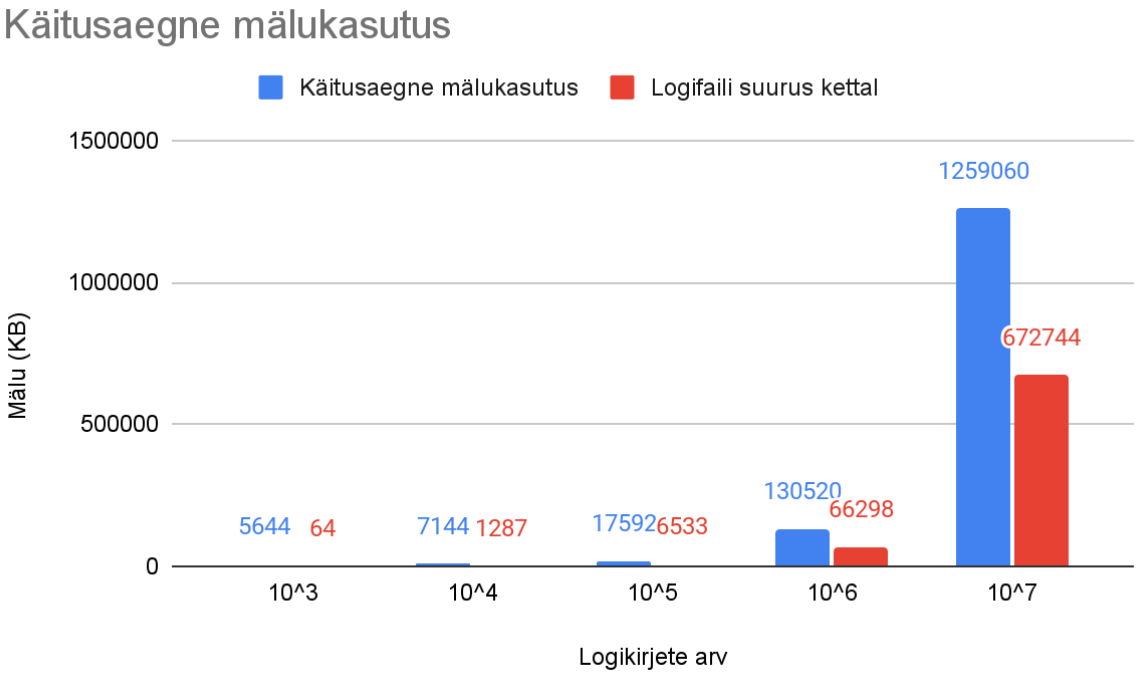
Alglaadimise aja mõõtmine on rakendusse sisseehitatud, salvestades hetkeaja programmi töö alguses ning väljastades kulunud aja peale logifailide esimest lugemist. Tulemustest selgus, et väikeste logifailide korral on alglaadimisele kuluv aeg kaduvväike ning kasvab lineaarselt sõltuvalt logifaili suurusest. See võimaldab rakendusel edukalt skaleeruda ka suurte andmehulkade korral (vt. Joonis 17).



Joonis 17. Rakenduse alglaadimiseks kulunud aeg.

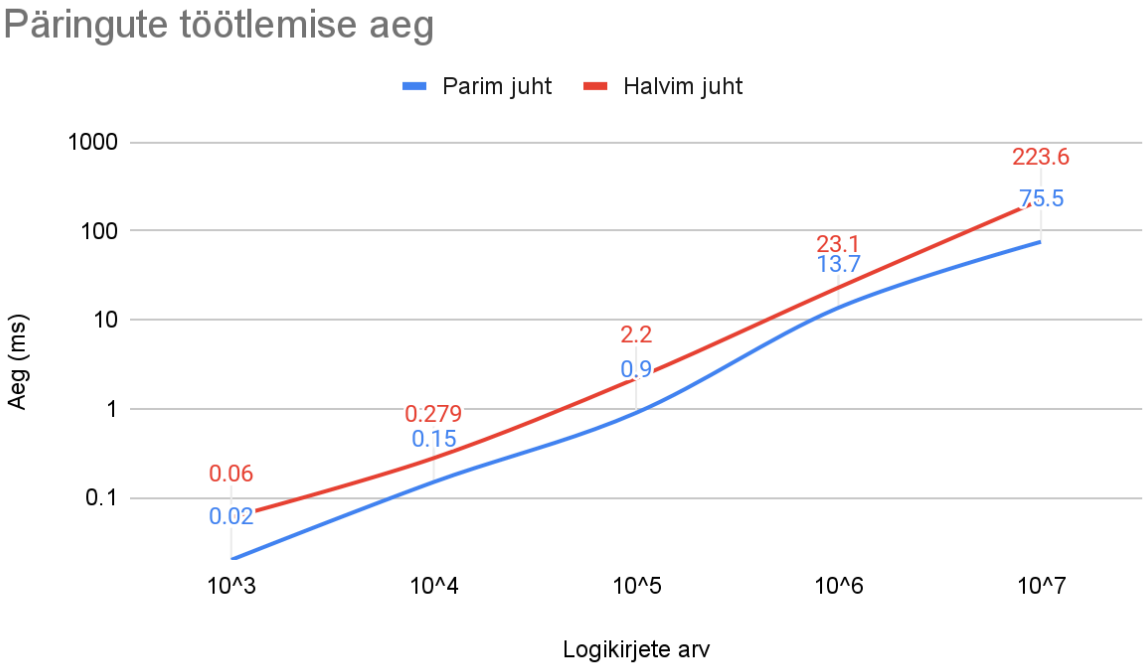
Käitusaegse mälukasutuse mõõtmiseks kasutati Windows Resource Monitor programmi. Täpsemalt Memory vaates asuvat Private tulpa, mis sisaldab rakenduse poolt kasutatavat mälu hulka, mida ei jagata ühegi teise rakendusega. Logifailide suurused leiti Windows Exploreri kaudu.

Tulemustest selgus, et väikeste logifailide puhul tuleb kasutajal arvestada tunduvalt suurema mälukasutusega võrreldes logifaili suurusega. See tuleneb asjaolust, et kogu rakendus ise vajab ~5.6 MB mälu. Logifaili suuruse kasvades muutub mälukasutus aga märgatavalt efektiivsemaks, nõudes 10 000 000 logikirje hoiustamiseks ~50% rohkem ruumi võrreldes faili suurusega kettal (vt. Joonis 18). Kasutajal on võimalik seda soovi korral veelgi vähendada, jättes parsimata näiteks logikirje allika.



Joonis 18. Rakenduse käitusaegne mälukasutus ja logifaili suurus kettal.

Päringute töötlemiseks kuluva aja leidmiseks kasutati sama metoodikat, mida alglaadimiseks kuluva aja leidmiseks. Aega mõõdeti parimal juhul, kui kasutaja pärib viimast 100 logikirjet ilma filtriteta ning halvimal juhul, kui kasutaja pärib logikirjeid filtriga, millele ei vastanud puhvris ükski logikirje. Tulemustest selgus, et päringute töötlemine on äärmiselt kiire isegi suurte logifailide puhul, kus 10 000 000 logikirje täielik filtreerimine võttis halvimal juhul vaid 223.6 millisekundit. Lisaks selgus, et ka päringute töötlemise aeg kasvab logifaili suurusest sõltuvalt lineaarselt (vt. Joonis 19.).



Joonis 19. Halvimal ja parimal juhul päringute töötlemiseks kulunud aeg.

3.4 Potentsiaalsed edasiarendused

Kuigi valminud veebirakendus täidab funktsionaalsuse poolest oma algset eesmärki, leidub mitu potentsiaalset edasiarendust ja parandust. Parandada võiks konfiguratsiooni protsessi, muuta parsimiseks kasutatava regulaaravaldise loomist lihtsamaks ja vähendada kompileeritud binaarfaili suurust.

Hetkel toimub rakenduse konfigureerimine peamiselt läbi TOML faili, mida tuleb teha tavalises tekstiredaktoris. See pole aga mugav juhtudel, kus rakendust kasutatakse üle võrgu ja konfiguratsiooni manuaalne muutmine on raskendatud. Seetõttu tuleks kasutajapoolsesse komponenti lisada uus vaheleht, mis võimaldaks kasutajal otse veebilehitsejas konfiguratsiooni muuta.

Suurimaks komistuskiviks kogu seadistus protsessi käigus kujuneb enamus kasutajatele tõenäoliselt parseri kirjutamine, kuna regulaaravaldised ei pruugi olla kõigile tuttavad. Seda probleemi on üritatud leevendada läbi näidisparseri konfiguratsiooni lisamisega, mille muutmiseks piisab vastavate nimegruppide ümber tõstmisest. Siiski saab seda protsessi veelgi

lihtsustada, koostades nimekirja populaarsematest logiformaatidest ning neile vastavatest regulaaravaldistest.

Valminud rakenduse binaarfailid on sõltuvalt platvormist suurusvahemikus 12.7 - 20 MB. Kuigi see on juba võrdlemisi väike suurus, saab seda kindlasti veelgi vähendada läbi mitmesuguste optimisatsioonide. Näiteks saab paremini läbi mõelda kliendipoolses komponendis kasutatavad lisad ja nende vajadus ning serveripoolses koodis eemaldada vähekasutatud sõltuvused.

Kokkuvõte

Töö eesmärgiks oli luua tavakasutajatele suunatud veebirakendus logide haldamiseks ning logimisraamistik programmeerimiskeeles Rust. Tavakasutajatele suunatud lahendused on tänapäeval muutumas üha aktuaalsemaks, kuna järjest rohkem inimesi soovib tellimuspõhiste pilveteenuste juurest liikuda ise hallatavate alternatiivide juurde.

Töö esimeses peatükis anti ülevaade Rust programmeerimiskeele olulisematest osadest. See võimaldab lugejal mõista töö praktilises osas loodud tarkvara arendamisel tehtud valikuid ning tutvustab Rust programmeerimiskeele tugevusi.

Seejärel anti põhjalik ülevaade logpeek nimelise logimisraamistiku loomisest. Toodi välja juba olemasolevate lahenduste puudused, kirjeldati detailselt logimisraamistiku ülesehitust ning põhjendati arenduse käigus tehtud valikuid. Valminud tarkvarateek võimaldab kasutajatel logija seadistada vaid kahe koodireaga ning parandab mitmeid olemasolevate lahenduste puuduseid.

Viimasena kirjeldati logpeek-server nimelise veebirakenduse loomise protsessi. Põhjendati loodava tarkavara vajadus, anti ülevaade projekti tehnilisest teostusest ning viidi läbi jõudlustestimine. Valminud tarkavara on kasutatav nii lokaalselt kui ka üle võrgu, toetab pea kõiki logide formaate ning on levitatav paigaldust mittevajava binaarfailina.

Jõudlustestimise tulemustest selgus, et rakendus on võimeline kiiresti töötlema suurel hulgal logikirjeid, nõudes 1 000 000 logikirje täielikuks töötlemiseks alla sekundi. Kasutaja päringutele vastamine on samuti kiire, vajades 1 000 000 logikirjest info otsimiseks vaid 23 millisekundit.

Töö edasiarendamiseks on mitmeid võimalusi. Kuna valminud veebirakendus on ennekõike suunatud tavakasutajatele, siis oleks kasulik muuta konfigureerimise protsessi veelgi lihtsamaks. Seda saab teha kasutajaliidesesse uue vahelehe lisamisega, mis võimaldaks konfiguratsiooni faili muuta ka üle võrgu.

Viidatud kirjandus

[1] Robert Lemos. Google, Microsoft Take Refuge in Rust Language's Better Security. https://www.darkreading.com/application-security/google-microsoft-take-refuge-in-rust-languages-better-security (23.04.2024)

[2] Press Release: Future Software Should Be Memory Safe.

https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report (23.04.2024)

[3] Perkel JM. Why scientists are turning to Rust. Nature. 2020;588(7836):185-186. doi:10.1038/d41586-020-03382-2

[4] Claburn, Thomas (2022-06-23). "Linus Torvalds says Rust is coming to the Linux kernel". The Register.

[5] Bugden, W., & Alahmar, A. (2022). Rust: The programming language for safety and performance. arXiv preprint arXiv:2206.05503.

[6] The Rust Standard Library. https://doc.rust-lang.org/std/index (02.12.2023)

[7] The Rust Programming Language. https://doc.rust-lang.org/book/title-page (27.12.2023)

[8] Async-await on stable Rust! https://blog.rust-lang.org/2019/11/07/Async-await-stable (01.01.2024)

[9] Asynchronous Programming in Rust. https://rust-lang.github.io/async-book (01.01.2024)

[10] Sara Verdi (2023). Why Rust is the most admired language among developers. The GitHub blog.

[11] Axum. https://docs.rs/axum/latest/axum/ (01.04.2024)

[12] Why Vite. https://vitejs.dev/guide/why.html (11.04.2024)

Lisad

I. Tarkvara lähtekood

Töö raames valminud logimisraamistiku lähtekood on leitav aadressil https://github.com/TheHighestBit/logpeek.

Veebirakenduse lähtekood koos kompileeritud binaarfailidega on leitav aadressil https://github.com/TheHighestBit/logpeek-server.

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Joosep Orasmäe**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   „**Logimisraamistiku ning veebirakenduse väljatöötamine programmeerimiskeeles Rust**”, mille juhendajad on **Varmo Vene** ja **Vambola Leping**, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Joosep Orasmäe

22.04.2024