



Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Marko Korhonen

Rust web-ohjelmointikielenä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja tietoviestintätekniikka

Insinöörityö

20.5.2020

Tekijä(t) Otsikko	Marko Korhonen Rust web-ohjelmointikielenä
Sivumäärä Aika	31 sivua + 6 liitettä 20.5.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja tietoviestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaaja(t)	Lehtori Simo Silander
<p>Tämän insinöörityön tavoitteena oli selvittää Rust-ohjelmointikielen soveltuvuutta web-ohjelmointiin.</p> <p>Insinöörityössä käydään läpi web-ohjelmoinnin perusteet sekä Rustin pääominaisuudet. Lukijalta oletetaan hyvin vähän tietämystä ja kaikki olennaiset käsitteet käydään läpi perusteista alkaen.</p> <p>Insinöörityön yhteydessä tehtiin projekti, missä palvelin- ja asiakaspuoli toteutettiin Rustilla. Projektin kaikki osa-alueet ja kaikki käytetyt riippuvaisuudet sekä syyt niiden valitsemiseen on käyty läpi perusteellisesti. Myös kehitysympäristön asentaminen ja projektin aloittaminen on käyty läpi alusta alkaen.</p> <p>Projektista kerätyn käytännön tiedon avulla on arvioitu Rustin soveltuvuutta web-ohjelmointiin erikseen sekä asiakas- että palvelinpuolella. Lopuksi on siirretty katsetta hieman tulevaan ja arvioitu, miten kielen soveltuvuus tulee todennäköisesti muuttumaan tulevaisuudessa. Lopussa on myös suosituksia Rustin sisällyttämisestä uusiin ja olemassa oleviin web-ohjelmointiprojekteihin.</p>	
Avainsanat	Rust, WebAssembly, ohjelmointi, web, full-stack

Author(s) Title	Marko Korhonen Rust as a Web Development Language
Number of Pages Date	31 pages + 6 appendices 20 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer
<p>The purpose of this thesis was to evaluate the suitability of the Rust programming language for web programming.</p> <p>Over the course of the report, both the basics of Rust and web programming are reviewed. The reader is expected to know only basic things, as all of the relevant concepts are explained in great detail.</p> <p>During the thesis, a project was carried out, in which both the frontend and backend was written in Rust. Based on this project, the suitability of Rust as a web programming language was analyzed. The setup of the development environment needed in developing Rust is also thoroughly explained.</p> <p>The practical knowledge acquired during the development of the project is applied to assessing the suitability of Rust as a web programming language. This is done separately concerning client side and server side. In the conclusion, the point of view is focused in the future and how this conclusion might change in the next few years. There's also practical tips on how one could apply Rust to new and existing web programming projects.</p>	
Keywords	Rust, WebAssembly, programming, web, full-stack

Sisällys

Lyhenteet

Sanasto

1	Johdanto	1
2	Web-ohjelmointi	2
2.1	Asiakaspuoli	2
2.2	Yhden sivun ohjelmat	2
2.3	Palvelinpuoli	3
3	Kielet	3
3.1	Rust	3
3.1.1	Muistinhallinta	4
3.1.2	Vahva tyyppitys	5
3.1.3	Muuttumaton data	7
3.1.4	Luotettavuus	7
3.1.5	Suorituskyky	8
3.1.6	Turvallinen rinnakkaisajo	8
3.1.7	Makrot	9
3.1.8	Dokumentaatio ja yhteisö	10
3.1.9	Paketinhallinta	10
3.2	WebAssembly	11
4	Projekti	12

4.1	Tavoitteet	12
4.2	Kehitysympäristön asennus	13
4.3	Palvelinpuoli	14
4.3.1	Kehys	14
4.3.2	Todentaminen	15
4.3.3	CORS	18
4.3.4	Tietokanta	19
4.4	Asiakaspuoli	20
4.4.1	Asennus	20
4.4.2	Kehys	21
4.4.3	Ulkonäkö	21
4.4.4	Reititys	22
4.5	Ongelmat	23
4.5.1	Evästeet	23
4.5.2	Tietokantayhteys	24
4.6	Tulokset	24
5	Rustin soveltuvuus web-ohjelmointiin	25
5.1	Palvelinpuoli	25
5.2	Asiakaspuoli	26
6	Yhteenveto	27
	Lähteet	29
	Liitteet	
Liite 1	Palvelinpuolen käynnistys (main.rs, entrypoint)	
Liite 2	JWT-tunnisteen luominen ja vahvistaminen	
Liite 3	Asiakaspuolen käynnistys (main.rs, entrypoint)	
Liite 4	Sisäänkirjautuminen	
Liite 5	Suojatun tiedon noutaminen	
Liite 6	Tietokannan migraatiotiedostot	

Lyhenteet

API	Application Programming Interface. Ohjelmointirajapinta.
CORS	Cross-Origin Resource Sharing. Mekanismi resurssien noutamiselle eri lähteistä.
CSS	Cascading Style Sheets. HTML-dokumenttien ulkonäön määrittämiseen tarkoitettu kieli.
HTML	HyperText Markup Language. Verkossa käytettävä tiedostomuoto.
HTTP	Hypertext Transfer Protocol. Alunperin HTML-dokumenttien välittämiseen kehitetty protokolla. Nykyään käytetään paljon muuhunkin.
JavaScript	Web-kehityksessä suosittu ohjelmointikieli.
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto, joka muistuttaa JavaScript-objekteja.
JWT	JSON Web Token. Avoin standardi käyttöoikeuksien hallinnoimiseen.
ORM	Object-relational mapping. Ohjelman objektien sovittamista relaatiotietokannan tauluihin ja tietokannasta takaisin ohjelmaan.
PHP	Hypertext Preprocessor. HTML-dokumenttien muokkaamiseen palvelimella käytetty ohjelmointikieli.
REST	Representational state transfer. Rajapintojen rakentamiseen tehty malli.
SOP	Same-origin policy. Turvallisuusmalli, missä asiakaspuolen ohjelman ei anneta noutaa resursseja muualta kuin samalta palvelimelta mistä ohjelma on noudettu.

Sanasto

Asiakaspuoli	Loppukäyttäjän käyttöliittymä ohjelmaan. Sanaa käytetään usein web-kehityksessä, jossa sillä viitataan nettiselaimeen.
Full-stack	Sovellus, missä on toteutettu sekä asiakas- että palvelinpuolen ohjelmisto.
Palvelinpuoli	Ohjelman palvelimella suoritettava osa, johon asiakaspuolen ohjelmisto yhdistyy.

1 Johdanto

Älylaitteiden ja tietokoneiden kehityksen sekä internetin parantuneen saatavuuden seurauksena verkon kautta käytettävien palveluiden kysyntä ja tarjonta ovat kasvaneet kovalla vauhdilla viime vuosina. Tästä johtuen koko internetin luonne on muuttunut täysin. Olemme todella kaukana 90-luvun dokumenttipohjaisesta internetistä. Nykyaikaiset nettisivut ovat täysverisiä ohjelmia.

Tämän takia myös web-ohjelmoinnissa käytettävissä teknologioissa on tapahtunut paljon muutoksia. Jotta nettisivuista saatiin pelkkien HTML-dokumenttien sijasta ohjelmia, 90-luvun puolivälissä julkaistiin PHP (palvelinpuolen kieli) ja JavaScript (asiakaspuolen kieli). Nykyään näistä kahdesta JavaScript on ehdottomasti suositumpi, minkä takia keskityn tässä insinööriyössä sen vertailuun Rustin kanssa.

Vaikka JavaScript onkin suosittu, siinä on paljon ongelmia [1]. Suurimmat ongelmat liittyvät kielen heikkoon tyyppitykseen. Se saattaa aiheuttaa ongelmia, jotka ilmenevät vasta ajon aikana ja vain tietyissä rajatapauksissa. Tämän takia koodin testaamiseen ja laadun ylläpitämiseen pitää panostaa enemmän. Toinen iso ongelma on se, että JavaScript ei ole tarkka koodin tyylistä. Jo pelkästään funktion kirjoittamiseen on useita eri tyyliä. Tämä hankaloittaa projekteissa kommunikointia ja työtoverien koodin lukemista.

Koska JavaScript on tulkittava kieli, se tulee aina häviämään suorituskäytössä käännettävälle kielille. Tätä eroa on osittain kavennettu ajonaikaisen kääntämisen optimoinneilla.

Edellä mainittujen syiden takia lähivuosina on ilmaantunut useita vaihtoehtoja sekä palvelin- että asiakaspuolen ohjelmointikieliksi. Näistä yksi itseäni eniten kiinnostava on Rust. Tämän insinööriyön yhteydessä toteutettu projekti käyttää Rustia sekä palvelimella että asiakaspuolella.

2 Web-ohjelmointi

Web-ohjelmoinnilla tarkoitetaan nettiselaimen välityksellä käytettävien ohjelmien ohjelmointia. Web-ohjelma jaetaan useimmiten kahteen osaan: asiakaspuoleen (eng. client side) ja palvelinpuoleen (eng. server side).

2.1 Asiakaspuoli

Asiakaspuoli koostuu käyttäjän nettiselaimesta, missä käyttöliittymän rakenne määritetään HTML-dokumentissa. HTML-dokumenttia voi muokata käyttämällä DOM:ia [2]. DOM, eli dokumenttiosiomalli on tapa kuvata HTML-dokumenttia puurakenteena, joka mahdollistaa dokumentin elementtien lukemisen ja muokkaamisen. Muokkaaminen suoritetaan asiakaspuolen web-ohjelmointikielellä, johon on tällä hetkellä kaksi vaihtoehtoa: JavaScript sekä WebAssembly, jota tarkastellaan tässä insinööriyössä tarkemmin.

Käyttöliittymän ulkonäköä ja toiminnallisuutta voi myös muokata palvelinpuolella. Tätä kutsutaan palvelinpuolen renderöinniksi ja siihen useimmiten käytetty ohjelmointikieli on PHP. Kun käyttöliittymän ulkoasua muokataan PHP:lla, HTML-dokumenttia muokataan palvelimella ennen kuin se lähetetään asiakkaalle. Palvelinpuolen renderöinti on kuitenkin jäänyt taka-alalle yhden sivun ohjelmien yleistyessä.

2.2 Yhden sivun ohjelmat

Yhden sivun ohjelmat [3] ovat tapa toteuttaa asiakaspuolen ohjelmia, missä selain lataa yhden ohjelman ja HTML-dokumentin, minkä jälkeen sivujen uudelleenlatauksia ei enää tapahdu. HTML-dokumentin DOM:ia muutetaan niin sanotusti lennosta. Tämä mahdollistaa sulavamman käyttökokemuksen asiakkaan näkökulmasta sekä helpottaa sovelluksen tilanhallintaa kehittäjän näkökulmasta.

Ohjelman tilaa hallinnoidaan usein lukemalla ja muuttamalla asiakkaan nettiselaimen osoitekenttää. Tätä kutsutaan asiakaspuolen reititykseksi. Reititys mahdollistaa myös tiettyyn ohjelman tilaan siirtymiseen suoraan, mikä on hyödyllistä esimerkiksi silloin, kun sovelluksen käyttäjillä on tarve jakaa linkkejä ohjelmaan.

2.3 Palvelinpuoli

Palvelinpuolella tarkoitetaan ohjelman osaa, johon asiakaspuoli yhdistyy noutaakseen tai tallettaakseen tietoa. Palvelinpuolen ohjelmointikielissä on huomattavasti enemmän vaihtoehtoja, sillä palvelin kommunikoi asiakkaan kanssa käyttämällä HTTP-protokollaa, johon löytyy kirjastoja lähes jokaiselle ohjelmointikielelle.

Palvelinpuoleen viitataan usein myös rajapintana (eng. Application Programming Interface, API), sillä tämänkaltaisia HTTP-protokollalla ohjattavia ohjelmia voi käyttää myös muuallakin kuin asiakaspuolen web-ohjelmissa. Rajapinnat voivat myös yhdistyä toisiinsa internetin välityksellä ja näin jakaa tietoja eri palveluiden välillä.

Rajapintojen yleistyessä on kehitetty standardeja, joiden mukaan rajapintoja voi rakentaa. Näin rajapinnoista tulee helpompia käyttää kaikille. Yksi näistä standardeista on Representational state transfer (REST) [4], jota pyrin käyttämään tämän insinööriyön projektia ohjelmoidessa.

3 Kielet

3.1 Rust

Rust [5] on Mozillan 2010 julkaisema ohjelmointikieli. Se on hyvin suorituskykyinen järjestelmätason ohjelmointikieli, joka muistuttaa monilta osin C- ja C++ -kieliä. Rustin tarkoituksena on säilyttää näiden vanhojen kielten suorituskyky, mutta kuitenkin tarjota samalla muun muassa vahva tyyppitys ja taattu turvallinen rinnakaisajo. Lisäksi tyypilliset C-kielen muistinhallintaongelmat on pyritty ratkaisemaan käytännöllä, jotka ovat samalla tehokkaita suorituskyvyn näkökulmasta, mutta myös helppoja käyttää ohjelmoijalle.

3.1.1 Muistinhallinta

Monissa korkean tason ohjelmointikielissä, esimerkiksi JavaScriptissä, on automaattinen roskienkeräys [6] (engl. garbage collector). Se on prosessi, joka siivoaa muistista käyttämättömiä tietoja ja näin ollen vapauttaa muistia. Automaattisen roskienkeräyksen ongelma on, että se itsessään käyttää järjestelmän resursseja. Lisäksi roskienkeruu on hidasta.

Automaattiselle roskienkeruulle on aikaisemmin ollut vaihtoehtona vain manuaalinen muistinhallinta, missä ohjelmoija varaa ja vapauttaa muistia tarpeen mukaan. Tämä on taas verrattuna automaattiseen roskien keruuseen melko työlästä ja viriheherkkää.

Rustin yhtenä pääominaisuutena on mainostettu sen uudenlaista näkökulmaa muistinhallintaan: omistajuutta [7]. Siinä jokaisella arvolla on omistaja, ja kun omistaja menee näkyvyysalueen ulkopuolelle, niin menevät myös sen omistamat arvotkin, eli ne vapautetaan muistista. Arvojen omistajuutta voi siirtää joko pysyvästi tai väliaikaisesti lainaamalla.

```
1 fn say_hello(name: String) {  
2     println!("Hello {}!", name);  
3 }  
4  
5 fn main() {  
6     let name = String::from("Marko");  
7     say_hello(name);  
8  
9     println!("{}", name);  
10 }
```

Koodiesimerkki 1: Omistajuus Rustissa

Koodiesimerkin 1 rivillä 9 tapahtuva "name"-arvon tulostus ei toimi, koska arvon omistajuus on siirretty funktiolle "say_hello". Kun funktio on suoritettu, se poistuu näkyvyysalueelta ja kaikki sen omistamat arvot vapautetaan muistista. Näin ollen arvoa "name" ei ole enää olemassa, kun sitä yritetään käyttää rivillä 9. Koodi saadaan toimimaan pienellä muutoksella.

```

1 fn say_hello(name: &String) {
2     println!("Hello {}!", name);
3 }
4
5 fn main() {
6     let name = String::from("Marko");
7     say_hello(&name);
8
9     println!("{}", name);
10 }

```

Koodiesimerkki 2: Linaus

Koodiesimerkissä 2 omistajuuden siirtäminen on korvattu lainauksella. Tämä muutos pitää tehdä sekä funktion parametrien määrittämiseen riville 1 että funktion kutsuun riville 7. Lainauksessa arvon omistajuus säilyy nykyisellään ja lainaaja antaa itse arvon sijasta viitteen (eng. reference). Viite on osoitin, joka osoittaa samaan muistipaikkaan kuin missä alkuperäinen arvo on. Lainaaminen tehdään käyttämällä merkkiä "&".

3.1.2 Vahva tyyppitys

Rust on vahvasti tyyppitetty kieli, mikä tarkoittaa sitä, että kaikkien arvojen tyytit pitää olla tiedossa ohjelman kääntämisen aikana. Tähän sisältyy myös funktioiden parametrit ja paluuarvot. Usein Rustin kääntäjä osaa päätellä (eng. inference) arvojen tyytit itse, varsinkin yksinkertaisissa tapauksissa.

```

1 let name = "Marko"; // Merkkijono
2 let age = 26;       // Kokonaisluku

```

Koodiesimerkki 3: Tyypin päättely

Niissä tapauksissa, joissa tyyppille voi olla useita vaihtoehtoja, tai silloin jos arvon määrittäksen yhteydessä tapahtuu konversio, ohjelmoijan tulee määrittää tyyppi.

```

1 let name: &str = "Marko";
2 let age: u8 = 26;

```

Koodiesimerkki 4: Tyypin merkintä

Koodiesimerkissä 4 valitsin arvolle "age" tyypin u8, koska se on pienin kokonaislukutyyppiä ja sen arvo voi olla välillä 0-255. Näin voi potentiaalisesti vähentää ohjelman muistin

käyttöä. Lisäksi tyypeillä voi karkeasti rajata funktion parametrien arvojen vaihteluväliä. Esimerkiksi jos on kirjoittamassa funktiota, joka ottaa parametrina henkilön iän, u8 on hyvä valinta, koska ihmisen ikä ei voi olla negatiivinen ja yksikään ihminen tuskin elää yli 255 vuotta.

Edellä mainittu tyyppi u8 on niin kutsuttu etumerkitön kokonaisluku (eng. **unsigned integer**). Etumerkillisissä kokonaisluvuissa (eng. **signed integer**) käytetään yksi bitti merkkamaan sitä, onko luku positiivinen vai negatiivinen. Etumerkittömissä luvuissa tätä ei tehdä, joten luku voi olla hieman isompi kuin etumerkillinen luku, mutta se ei voi olla negatiivinen.

Etumerkitön			Etumerkillinen		
Tyyppi	Minimi	Maksimi	Tyyppi	Minimi	Maksimi
u8	0	$2^8 - 1$	i8	-2^7	$2^7 - 1$
u16	0	$2^{16} - 1$	i16	-2^{15}	$2^{15} - 1$
u32	0	$2^{32} - 1$	i32	-2^{31}	$2^{31} - 1$
u64	0	$2^{64} - 1$	i64	-2^{63}	$2^{63} - 1$
u128	0	$2^{128} - 1$	i128	-2^{127}	$2^{127} - 1$

Taulukko 1: Rustin kokonaislukutyypit ja niiden vaihteluvälit

Toisin kuin C- ja C++ -kielissä, Rustissa on oletuksena etumerkittömien kokonaislukujen ylivuoto pois päältä. Tämä tarkoittaa sitä, että jos esimerkiksi 8-bittisen etumerkittömän kokonaisluvun (u8) arvoksi yritetään asettaa 256, siitä tulee 0. Rustin kääntäjä siis ei anna tällaisen tapahtua vaan kääntämisen yhteydessä tulee virheviesti, jonka voi nähdä koodiesimerkissä 5.

```
error: literal out of range for `u8`
--> types.rs:2:19
  |
2 |     let age: u8 = 256;
  |                   ^^^
  |
  = note: `#[deny(overflowing_literals)]` on by default

error: aborting due to previous error
```

Koodiesimerkki 5: Etumerkittömän kokonaisluvun ylivuoto

3.1.3 Muuttumaton data

Rustissa kaikki arvot ovat oletuksena muuttumattomia (engl. immutable). Jos muuttumattoman datan sijasta tarvitsee muuttujia (engl. mutable), voi käyttää avainsanaa "mut", esimerkiksi `let mut name = "Marko"`. Myös lainaukset suoritetaan oletuksena muuttumattomasti ja muutettavan lainauksen voi tehdä samalla avainsanalla, esimerkiksi `say_hello(&mut name)`

Yleinen konsensus ohjelmoinnin maailmassa on se, että kaikki arvot, joita ei tarvitse muuttaa, pitäisi nimenomaan määrittää muuttumattomina. Tämä on tuttua kaikille, jotka ovat tutustuneet funktionaalisiin ohjelmointikieliin. Muuttumaton data on myös todella tärkeää rinnakkaisajossa, missä useampi prosessi suorittaa samoja funktioita ja käsittelee samoja arvoja samaan aikaan.

3.1.4 Luotettavuus

Rustia kehitettäessä on aina ollut tavoitteena luotettavuus. Tämä tarkoittaa sitä, että ohjelman virheet huomataan jo kääntämisen yhteydessä, eikä vasta suorituksen aikana ohjelman tietyssä tilassa. Tämän mahdollistavat edellä mainitut omistajuusmalli ja vahva tyyppitys. Omistajuusmalli varmistaa sen, että ohjelmoija joutuu koodia kirjoittaessaan miettimään arvojen eliniän, joka tekee muistivuoista harvinaisia. Vahva tyyppitys varmistaa taas sen, että data on kaikkialla ohjelmassa yhteensopivaa.

Myös Rustin kääntäjään on panostettu paljon, ja virheiden sattuessa se on todella hyvä työkalu ohjelmoijalle. Se alleviivaa ongelmakohtia ja selittää lyhyesti, mistä ongelma johtuu. Jossain tapauksissa kääntäjä jopa antaa pieniä koodin pätkiä, mistä voi olla apua ongelman ratkaisemisessa [koodiesimerkki 6].

```

error[E0412]: cannot find type `Json` in this scope
--> src/component/login.rs:6:16
  |
6 | fn create() -> Json {}
  |               ^^^^^ not found in this scope
  |
  | help: possible candidates are found in other modules, you can import
  |       them into scope
1 | use yew::format::Json;
  |
1 | use yew::format::json::Json;
  |

```

Koodiesimerkki 6: Rustin kääntäjä auttaa unohtuneen importin lisäämisessä

3.1.5 Suorituskyky

Rustin monista luotettavuuteen liittyvistä ominaisuuksista johtuen voisi luulla, että kaikki nämä ominaisuudet vaikuttaisivat negatiivisesti suorituskykyyn. Monissa kielissä onkin runtime-kirjasto [8]. Se on kääntäjän jatke, joka lisää käännettyyn binääritiedostoon ajon aikana suoritettavia käskyjä, jotka liittyvät juuri tällaisten luotettavuusominaisuuksien taakamiseen. Nämä käskyt suoritetaan siis itse ohjelman suorituksen yhteydessä, joten ne käyttävät prosessoriaikaa ja muistia aina, kun ohjelma on käynnissä.

Rustissa ei ole runtime-kirjastoa. Kääntäjä tekee kaikki runtime-kirjaston tehtävät kääntämisen yhteydessä. Jos jokin tarkistus ei mene läpi, esimerkiksi omistajuusmallia on käytetty väärin, ohjelman kääntäminen keskeytetään ja kehittäjälle näytetään virheviesti.

Runtime-kirjaston puuttuminen vapauttaa prosessointiaikaa ja muistia itse ohjelman käyttöön, mikä parantaa suorituskykyä.

3.1.6 Turvallinen rinnakkaisajo

Rinnakkaisajo tarkoittaa sitä, että jotakin ohjelman osaa suoritetaan saman aikaan useassa prosessorin säikeessä. Rinnakkaisajolle sopivissa tehtävissä kuorman jakamisella saavutetaan parempaa suorituskykyä.

Turvallisella rinnakkaisajolla [9] tarkoitetaan sitä, että jos rinnakkaisajossa käsitellään tietorakenteita, käsittely suoritetaan turvallisesti. Turvallisuus tarkoittaa sitä, että saavutetaan haluttu tulos eikä ilmene sivuvaikutuksia.

Rust takaa turvallisen rinnakkaisajon luvussa 3.1.4 mainituilla luotettavuusominaisuuksilla. Tämän asian perinpohjaiseen selittämiseen en ryhdy tässä insinöörityössä. Asiasta kiinnostuneet voivat lukea asiasta lisää Rust Blogista [10].

3.1.7 Makrot

Yksi todella mielenkiintoinen ominaisuus Rustissa on makrot. Se on toiminnallisuus, mikä mahdollistaa metaohjelmoinnin [11]. Metaohjelmoinnissa koodia voi generoida kääntämisen aikana, mikä on erityisen hyödyllistä esimerkiksi silloin, kun ohjelmoija tarvitsee useita toiminnallisuudeltaan samankaltaisia funktioita.

```

1 macro_rules! laske {
2     (lisää $num1:literal ja $num2:literal) => {
3         println!("{}", plus {} on {}", $num1, $num2, $num1 + $num2);
4     };
5     (vähennä $num1:literal ja $num2:literal) => {
6         println!("{}", miinus {} on {}", $num1, $num2, $num1 - $num2);
7     };
8     (kerro $num1:literal ja $num2:literal) => {
9         println!("{}", kertaa {} on {}", $num1, $num2, $num1 * $num2);
10    };
11    (jaa $num1:literal ja $num2:literal) => {
12        println!("{}", jaettuna {} on {}", $num1, $num2, $num1 / $num2);
13    };
14 }
15
16 laske!(lisää 10 ja 269);
17 laske!(vähennä 652 ja 3);
18 laske!(kerro 256 ja 2);
19 laske!(jaa 100 ja 50);
20
21 // Tuloste:
22 // 10 plus 269 on 279
23 // 652 miinus 3 on 649
24 // 256 kertaa 2 on 512
25 // 100 jaettuna 50 on 2

```

Koodiesimerkki 7: Runsassanainen laskin toteutettuna Rustin makrona

Makron sisällä suluissa olevat lauseet ovat verrattavissa Rustin "match"-lauseeseen. Kun makron syöte vastaa jotakin näistä lauseista, hakasulkujen sisällä oleva koodi generoidaan. Koodiesimerkin 7 makrossa käytetty "println!()" on myös itsessään makro, joka tulee Rustin "std"-kirjaston mukana.

```
1 macro_rules! println {  
2     () => { ... };  
3     ($($arg:tt)*) => { ... };  
4 }
```

Koodiesimerkki 8: Rustin sisäänrakennettu println!() makro [12]

Metaohjelmointi avaa aivan uudenlaisia mahdollisuuksia sille, mitä ohjelmointikielellä voi tehdä. Makroilla voi toteuttaa vaikka kokonaisen ohjelmointikielen [13].

3.1.8 Dokumentaatio ja yhteisö

Rust on tunnettu todella laajasta dokumentaatiostaan ja vahvasta yhteisöstään. Molemista on paljon apua varsinkin aloittelijoille.

Aloitin itsekkin opiskelemaan Rustia vain hieman ennen tämän insinööurityön alkua. Yhteisöstä oli monesti apua projektin aikana vastaan tulleissa ongelmissa.

3.1.9 Paketinhallinta

Rustin paketinhallinta on toteutettu Cargo-nimisellä ohjelmalla. Sitä voi käyttää koko ohjelmiston elinkaaren ajan aina projektin luomisesta sen julkaisemiseen. Cargon käsittelemiä paketteja kutsutaan laatikoiksi (eng. crate), jotka julkaistaan crates.io-pakettirekisterissä [14]. Laatikot voivat myös olla riippuvaisia toisista laatikoista. Laatikon tiedot ja riippuvuudet määritetään Cargo.toml-tiedostossa [koodiesimerkki 9].


```

1  [package]
2  name = "thesis-backend"
3  version = "0.1.0"
4  authors = ["Marko Korhonen <marko.korhonen@reeky.net>"]
5  edition = "2018"
6
7  [dependencies]
8  actix-web = "2.0.0"
9  actix-rt = "1.0.0"
10 serde = { version = "1.0.104", features = ["derive"] }
11 diesel = { version = "1.4.3", features = ["mysql", "r2d2", "chrono"] }
12 dotenv = "0.15.0"
13 bcrypt = "0.6.2"
14 env_logger = "0.7.1"
15 r2d2 = "0.8.8"
16 crypto = "0.0.2"
17 jsonwebtoken = "7.1.0"
18 chrono = { version = "0.4.11", features = ["serde"] }
19 actix-cors = "0.2.0"
20 actix-identity = "0.2.1"
21 futures = "0.3.4"
22 actix-files = "0.2.1"

```

Koodiesimerkki 9: Projektin palvelinpuolen Cargo.toml

Cargo on saatavilla myös useita liitännäisiä, esimerkiksi cargo-watch, joka suorittaa halutun toiminnon aina, kun projektin sisällä tapahtuu muutoksia sekä tässäkin insinööri-työssä käytetty cargo-web, joka helpottaa WebAssembly-ohjelmien kehittämistä.

3.2 WebAssembly

WebAssembly [15] on kehitteillä oleva asiakaspuolen ohjelmointikieli. Sitä on suunniteltu JavaScriptin seuraajaksi ja sen suurimpana etuna verrattuna JavaScriptiin on huomattavasti matalamman tason esitysmuoto, minkä ansiosta se on suoritussykyisempi.

Kehittäjän ei ole tarkoitus kirjoittaa WebAssemblya itse, vaan käyttää työkaluja, joilla olemassa olevia ohjelmointikieliä voi kääntää WebAssemblyksi. Rust on tästä hyvä esimerkki, sillä WebAssembly on yksi sen kääntäjän natiiveista "targeteista", samalla tavalla kuin vaikka x86-prosessorit.

WebAssembly on ensisijaisesti binääriformaatti, mutta sen voi muuntaa myös tekstiformaatiksi, jonka nimi on WebAssembly text [16]. WebAssembly text käyttää syntaksissaan S-lausekkeita [17]. Se on notaatio puurakenteiselle datalle, joka on kehitetty Lisp-ohjelmointikieltä varten, joten WebAssembly text muistuttaa syntaksiltaan hyvin paljon Lispä. WebAssembly tekstiä käytetään tilanteissa, joissa ihmisen täytyy ymmärtää, mitä koodissa tapahtuu. Tätä hyödynnetään esimerkiksi WebAssemblyn sisäisessä kehityksessä ja web-ohjelmistojen debuggereissa.

```

1 (module
2   (func $plus (param $num1 i32) (param $num2 i32) (result i32)
3     local.get $num1
4     local.get $num2
5     i32.add)
6   (export "plus" (func $plus)))
7 )

```

Koodiesimerkki 10: Lukujen yhteenlaskufunktio WebAssembly text -formaatissa

```

1 fn plus(num1: i32, num2: i32) -> i32 {
2   num1 + num2
3 }

```

Koodiesimerkki 11: Lukujen yhteenlaskufunktio Rustilla

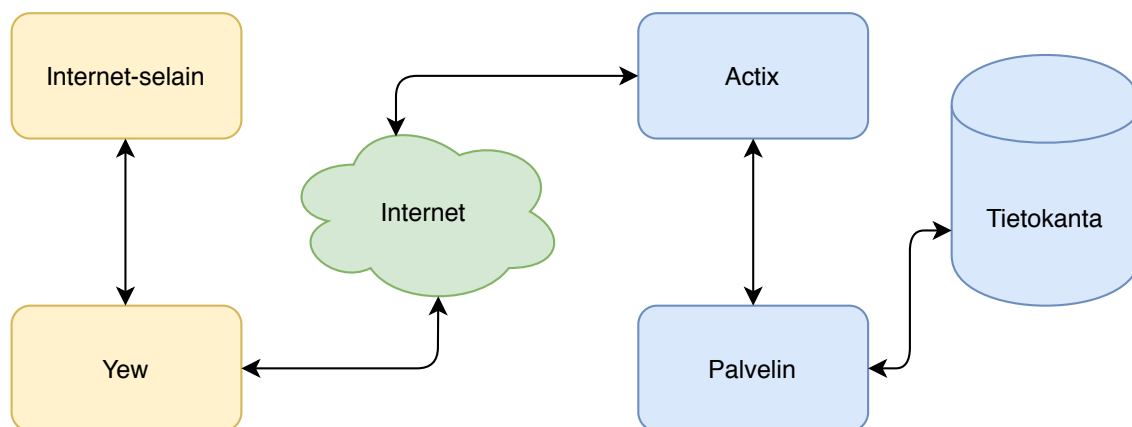
Koodiesimerkeistä 10 ja 11 voi nähdä saman koodin sekä Rustilla toteutettuna että WebAssembly text -formaatissa. Esimerkeistä huomaa hyvin WebAssemblyn matalemmän esitysmuodon käytännössä.

4 Projekti

Tein insinööriyön yhteydessä full-stack-projektin, jossa sekä palvelin- että asiakaspuolen ohjelmointi tehtiin Rustilla.

4.1 Tavoitteet

Tavoitteena ei ollut saada aikaiseksi mitään todella monimutkaista ohjelmaa, vaan puhtaasti arvioida Rustin soveltuvuutta web-ohjelmointiin yksinkertaisella esimerkkiprojektilla.



Kuva 1: Projektin arkkitehtuurimalli

Kuvassa 1 näkee sovelluksen arkkitehtuurin. Sovelluksen asiakaspuoli koostuu mallin keltaisista laatikoista ja palvelinpuoli sinisistä. Nuolet merkitsevät datan liikkumista, mikä on kaikkialla kaksisuuntaista. Yew ja Actix ovat valitsemani asiakas- ja palvelinpuolen sovel-luskehukset.

4.2 Kehitysympäristön asennus

Rust-projektin aloittamiseksi kehittäjä tarvitsee koneelleen Rustin paketinhallintatyökalun, Cargon (katso luku 3.1.9). Olen itse Linux-käyttäjä, joten sain asennettua Cargon Linux-jakeluni ohjelmavarastosta. Mac- ja Windows-käyttäjille suosittelen Rustup-asennusohjelman käyttämistä. Lisää tietoa Rustin asentamisesta saa Rustin kotisivuilta [18].

Projektin saa initialisoitua komennolla `cargo init projektinnimi`. Tämä komento luo hakemiston "projektinnimi", minkä sisällä on Cargon konfiguraatiotiedosto `Cargo.toml`, jossa voi määrittää projektin tiedot ja riippuvuudet.

Lähdekoodi sijaitsee hakemistossa "src". Cargo kirjoittaa hakemistoon valmiiksi "main.rs"-tiedoston, jossa on "Hello world!" -esimerkkikoodi. "main.rs" on aina Rust-ohjelman ensimmäiseksi suoritettava tiedosto, eli niin kutsuttu entrypoint. "main.rs"-tiedoston sisällä pitää olla "main()" -funktio, josta ohjelman suoritus alkaa. Projektin voi suorittaa komennolla `cargo run`.

Suosittelen asentamaan myös muutaman Cargon liitännäisen helpottamaan kehitystä. cargo-watchin avulla voi ajaa komennon aina, kun lähdekoodi muuttuu. Esimerkiksi komennolla `cargo watch -x run` voi kääntää ja käynnistää projektin aina uudelleen, kun lähdekoodi muuttuu. cargo-add-liitännäisellä voi helposti lisätä uusia riippuvuuksia projektiinsa. Esimerkiksi Actix webin saa lisättyä komennolla `cargo add actix-web`.

Näillä ohjeilla pääsee alkuun palvelinpuolen kehityksessä. Asiakaspuolen kehityksen käynnistäminen vaati vielä muutaman lisävaiheen. Siitä lisää luvussa 4.4.1.

4.3 Palvelinpuoli

4.3.1 Kehys

Palvelinpuolen kehykseksi valikoitui Actix web. Se on käytännössä vastine JavaScript-maailman Express.js:lle, eli se hoitaa HTTP-palvelimen työtä ja reitittää GET- ja POST-pyynnöt ohjelman oikeille funktioille.

Actix web on puolestaan rakennettu hyödyntämällä Actix-sovelluskehystä, joka on rakennettu löyhästi actor-mallin pohjalta. Actor-malli [19] on Carl Hewittin vuonna 1973 luoma matemaattinen ja tietotekninen malli rinnakkaisajosta. Tämän ansiosta Actix web on hyvin suorituskykyinen ja helposti skaalautuva ratkaisu rajapintoja rakennettaessa.

Actix web on myös hyvin integroitu Rustin vahvaan tyyppitykseen. Erityisen vaikuttavaa oli se, että esimerkiksi tietyn rajapinnan päätepisteen POST-pyyntöön pystyy määrittämään parametriksi tietyn rakenteen (eng. structure) [koodiesimerkki 12].

```

1 struct RegisterUser {
2     username: String,
3     password: String,
4     admin: bool,
5     password_confirmation: String,
6 }
7
8 #[post("/auth/register")]
9 fn register(new_user: actix_web::web::Json<RegisterUser>) ->
10     Result<HttpResponse, HttpResponse> {
11     register(new_user);
12 }

```

Koodiesimerkki 12: Rakenne POST-pyyntöön parametrinä

Tällöin varmistetaan automaattisesti siitä, että kun asiakkaan POST-pyyntö saapuu tähän funktioon, kaikki data on oikeanlaista tyyppiä ja kaikki tarvittava data on pyynnössä mukana. Verrattuna JavaScriptiin, jossa ei ole tyypejä, kaikki vastaanotettava data pitäisi tarkastaa käsin.

```

1 {
2     "username": "TestUser",
3     "password": "verysecurepassword",
4     "admin": 3
5 }

```

Koodiesimerkki 13: Rakenteeseen sovitettava JavaScript-objekti

Kun koodiesimerkin 12 funktioon "register" saapuu koodiesimerkin 13 mukainen JSON-objekti, Actix huomaa väärän tyyppin ja vastaa statuskoodilla "400 Bad Request", koska objektin parametri "admin" ei ole tyyppiä `bool`.

Palvelinpuolen initialisointi ja käynnistäminen löytyvät liitteestä 1.

4.3.2 Todentaminen

Actix webin modulaarisuus mahdollisti myös tarvittavien väliohjelmistojen (eng. middlewa-re) sisällyttämisen ohjelman toimintaan. Actix identity -paketista löytyi tarvittavat palikat, joilla sain lisättyä itse kirjoittamani käyttäjän todentamistoiminnallisuuden suojaamaan ha-luttuja reittejä.

Todentamiseen päätin käyttää JSON Web Tokeneita. Ne ovat standardoitu (RFC 7519 [20]) tunnistautumistapa verkossa. Tokenit ovat merkkijonoja, jotka sisältävät JavaScript-objekteja tekstimuodossa (JSON). Token koostuu kolmesta osasta [20]: ylätunnisteesta (eng. header), hyötykuormasta (eng. payload) ja allekirjoituksesta (eng. signature). JSON Web Tokenin kaikki osat on havainnollistettu kuvassa 2.

Encoded
PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJusIm5hbWUiOiJUZXR0VXNlciIsImFkbWludIjpmYWxzZSwiZXhwIjoxNTg5MTE3MDg0fQ.WHxzK1RFNjMwMQWP6-RA4QmZjsmnlcIXbwL8z0sKPQ
```

Decoded
EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "sub": 5,
  "name": "TestUser",
  "admin": false,
  "exp": 1589117084
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  po0IefEvoo0IefaiNai7uv
) secret base64 encoded
```

Signature Verified

SHARE JWT

Kuva 2: JWT:n osat havainnollistettuna käyttämällä jwt.io-sivustoa [20]

Ylätunniste koostuu JSON-objektista [koodiesimerkki 14], missä on määritelty tokenin tyyppi (tässä tapauksessa "JWT") ja allekirjoituksen algoritmi, mikä voi olla joko HS256 tai RSA.

```

1 {
2   "typ": "JWT",
3   "alg": "HS256"
4 }
```

Koodiesimerkki 14: JSON Web Tokenin ylätunniste

Tokenin toisessa osassa, hyötykuormassa, on itse tokenin sisältö [koodiesimerkki 15]. Tokenin sisällön voi päättää kokonaan itse, vaikkakin joitakin standardeja kenttiä on määritetty. Esimerkiksi iss (issuer), sub (subject) ja exp (expiration time). Päätin sisällyttää myös tiedon siitä, onko käyttäjä ylläpitäjä, koska tätä tietoa voi sitten käyttää asiakaspuolella esimerkiksi käyttöliittymän muokkaamiseen käyttäjän roolin perusteella. Usein myös

käyttäjän nimi sisällytetään tokeniin. Tokenin sisältöä suunnitellessa kannattaa pitää mielessä, että sen sisältö on nähtävissä kaikille, joten se ei ole oikea paikka tallettaa salaista tietoa, kuten vaikka käyttäjän salasana.

```

1 {
2   "sub": 5,
3   "name": "TestUser",
4   "admin": false,
5   "exp": 1588262665
6 }
```

Koodiesimerkki 15: Yhden JSON Web Tokenin hyötykuorma tästä projektista

Tokenin kolmesta osasta viimeinen on allekirjoitus. Se on koostettu ylätunnisteesta [koodiesimerkki 14] määritellyillä algoritmilla käyttämällä parametreina tokenin hyötykuormaa [koodiesimerkki 15] ja vain palvelimen tiedossa olevaa salasanaa. Koko tokenin turvallisuus perustuu juuri tähän allekirjoitukseen. Sen avulla palvelin voi varmistua siitä, että tokenia ei ole muokannut kukaan, kenellä ei ole tätä salasanaa.

Tokenia varmennettaessa hyötykuorma allekirjoitetaan uudelleen ja tätä uutta allekirjoitusta verrataan tokenin mukana tulleeseen allekirjoitukseen. Jos ne ovat samat, palvelin voi olla varma siitä, että tokenin sisältöön voi luottaa. Tämän edellytyksenä tietysti on, että salasanaa on säilytetty turvallisella tavalla.

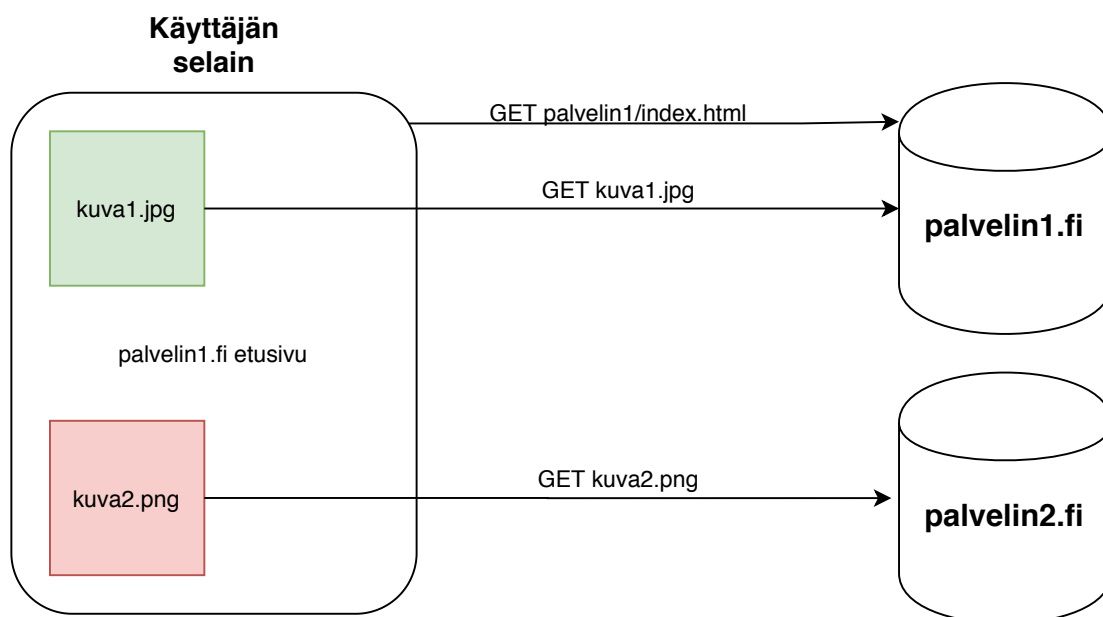
Otetaan esimerkkinä tämän sovelluksen käyttäjä Pasi. Pasi ei ole ylläpitäjä, mutta hän koittaa tehdä itsestään ylläpitäjän muuttamalla omasta tokenistaan parametrin `"admin": false` arvoksi `true`. Kun Pasi lähettää tämän muokatun tokenin palvelimelle, se viedään JWT-validointifunktioon. Palvelin huomaa, että token ei ole enää validi, koska sen sisältöä ei ole allekirjoitettu palvelimen salasanalla.

Päätin tallettaa edellä mainitun JSON Web Tokenin evästeeseen (eng. cookie), joka on standardi tapa tallettaa juuri tällaisia todentamiseen käytettäviä tietoja selaimissa. Evästeiden käyttämisen etu on se, että selain huolehtii sen tallettamisesta automaattisesti ilman lisätoimia kehittäjältä. Lisäksi selain sisällyttää sen seuraaviin kutsuihin automaattisesti.

Projektin JWT-toteutus on liitteessä 2.

4.3.3 CORS

Lisäsin palvelimelle myös Cross-Origin Resource Sharing (CORS) [21] -toiminnallisuuden. Oletuksena selaimen lataama ohjelma saa ladata resursseja vain samasta osoitteesta, kuin mistä itse ohjelma on ladattu. Tämä on turvallisuuskäytäntö, joka kulkee nimellä Same-origin policy (SOP) [22]. Näihin resursseihin sisältyvät muun muassa CSS-tyylimäärittelyt, kuvat ja JavaScript-ohjelmat.



Kuva 3: Esimerkki Single origin policy -turvallisuuskäytännöstä

Kuvassa 3 käyttäjä navigoi osoitteeseen "palvelin1.fi", josta selain lataa HTML-dokumentin. Tässä dokumentissa on kaksi kuvaa, "kuva1.jpg" ja "kuva2.png". Kuva 1 tulee samalta palvelimelta kuin mistä dokumenttikin, joten selain sallii sen lataamisen SOP-käytännön mukaisesti. Kuva 2 taas tulee toiselta palvelimelta, joten selain estää sen lataamisen.

Lähes kaikki nykyiset web-ohjelmistot kuitenkin vaativat näiden ulkoisten resurssien käyttöä. CORS on tekniikka, jossa palvelin kertoo selaimelle sallitut osoitteet, mistä resursseja saa noutaa.

Jotta kuvan 2 lataaminen saataisiin toimimaan, palvelimen tulee implementoida CORS-tekniikka. Palvelimen tulee lähettää asiakkaalle HTML-dokumentin yhteydessä ylätunniste [koodiesimerkki 16], joka sisältää palvelimen 2 osoitteen. Näin käyttäjän selain tietää, että palvelin 1 sallii resurssien noutamisen palvelimelta 2.

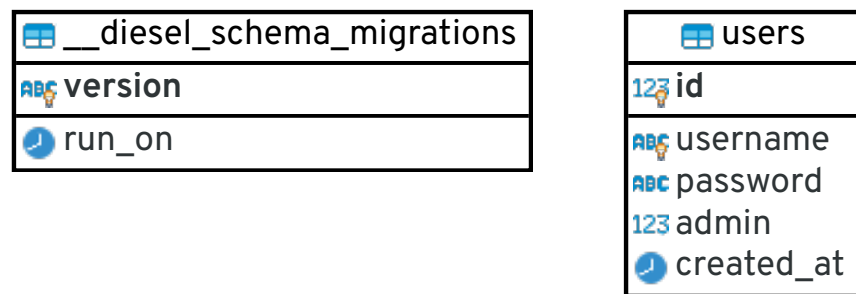
1 Access-Control-Allow-Origin: http://palvelin2.fi

Koodiesimerkki 16: Esimerkki CORS-ylätunnisteesta

CORS:n lisääminen tähän projektiin hoitui Actixin liitännäisellä `actix_cors`. CORS:n asetusten määrittäminen löytyy liitteestä 1.

4.3.4 Tietokanta

Tietokannaksi valikoitui itselleni tuttu MySQL. Koska projekti on yksinkertainen todennusdemo, tietokantaan tuli vain yksi taulu. Users-taulu sisältää käyttäjän tiedot: käyttäjänimen, salatun salasanan, tiedon käyttäjän ylläpitäjyydestä sekä käyttäjän luomisajankoh-
ta. Tietokannan ER-mallin näkee kuvasta 4.



Kuva 4: Tietokannan ER-malli

Relaatiotietokannan sai helposti yhdistettyä Rust-koodiini Diesel ORM-kirjastolla. Diesel on tähän mennessä käyttämistäni ORM-kirjastoista selkeästi mukavin käyttää.

Käytännöllisimmät ominaisuudet kehittäjän näkökulmasta olivat Dieselin mukana tuleva komentorivikäyttöliittymä ja migraatiot. Jokaiselle taululle luodaan uusi migraatio, esimerkiksi `diesel migration generate users`, jonka jälkeen Dieselin luomaan hakemistoon kirjoitetaan `up.sql`- ja `down.sql`-tiedostot [liite 6], eli ohjeet siitä, miten tämä taulu luodaan ja poistetaan. Taulu viedään tietokantaan komennolla `diesel migration run` ja taulun voi poistaa ja luoda uudelleen komennolla `diesel migration redo`. Tämä mahdollistaa myös samalla sen, että versiohallintaan voi tallentaa useita versioita samasta taulusta ja palata helposti takaisin vanhempaan versioon, jos uudemman kanssa ilmenee ongelmia.

Edellä mainitut työkalut helpottivat tietokannan kehitystä huomattavasti. Usein varsinkin projektin alkuvaiheilla tietokanta muuttuu jatkuvasti ja usein tulee tarve poistaa ja luoda tietokanta uudelleen. Monet kehittäjät pitävät juurikin tällaista Dieselin `up.sql:n` kaltaista tiedostoa versiohallinnassa ja tarpeen mukaan poistavat tietokannan käsin ja liittävät komennon tiedostosta tietokannan komentorivikäyttöliittymään. Dieselä käytettäessä tämä tulee tehtyä automaattisesti ja se tuntuu todella luontevalta.

Päätin myös käyttää Dieselin kanssa yhteyksien yhdistämistä (eng. connection pooling). Tämä tarkoittaa sitä, että palvelin luo käynnistyessään prosessin, joka ottaa yhteyden tietokantaan. Tämä prosessi puolestaan jakaa tätä yhteyttä sitä tarvitseville ohjelman osille. Tämä on tehokkaampaa kuin vaihtoehto, jossa otetaan uusi yhteys jokaista tietokantakyselyä varten.

4.4 Asiakaspuoli

4.4.1 Asennus

Projektin asiakaspuolen aloittaminen vaati aika paljon tutkimustyötä. Rust-koodi pitää kääntää WebAssemblyksi, johon on olemassa useita eri työkaluja. Lisäksi selaimien rajapintojen käyttämiseen tarvitsee jonkinlaisen kirjaston, joita Rust-maailmassa on tällä hetkellä kaksi: `stdweb` [23] ja `web_sys` [24]. Näistä `stdweb` on vanhempi, mutta myös tuorempi. `web_sys` on uudempi tulokas ja näyttää siltä, että se tulee korvaamaan `stdweb:n` tulevaisuudessa.

Valitsemani sovelluskehys tukee molempia kirjastoja, mutta päädyin kuitenkin valitsemaan `stdweb:n`. Suurimpana syynä oli projektin aloittamisen helppous, joka tehdään Cargon liitännäisellä `cargo-web`. Toisena syynä oli, että WebAssembly-ohjelman paketointi `web_sys:n` kanssa on tätä kirjoittaessani riippuvainen NodeJS:stä, kun taas `stdweb` ei vaadi NodeJS:n asennusta kehittäjän koneelle ollenkaan.

WebAssembly-koodin suorittamiseksi selain tarvitsee pienen pätkän JavaScriptiä, joka tekee tarvittavat toimet WebAssembly-ohjelman käynnistämiseksi. Tämän koodin generoi puolestani `cargo-web`. Lisäksi JavaScriptin suorittamiseen tarvitaan yksi HTML-tiedosto. Tähän tiedostoon voi myös sisällyttää metadataa, kuten esimerkiksi sivuston otsikon, jo-

ka näkyy selaimen välilehdessä. Tähän tiedostoon linkitetään myös kaikki muu staattinen data, kuten tyylimääritykset. Nämä tiedostot laitoin käytännön mukaisesti static-nimiseen hakemistoon asiakaspuolen projektin juureen. Tämän hakemiston linkitin symbolisella linkillä palvelinpuolen projektiin, jonka HTTP-serveri voi sitten lähettää HTML-dokumentin, JavaScript-tiedoston ja WebAssembly-binäärin käyttäjän selaimelle.

4.4.2 Kehys

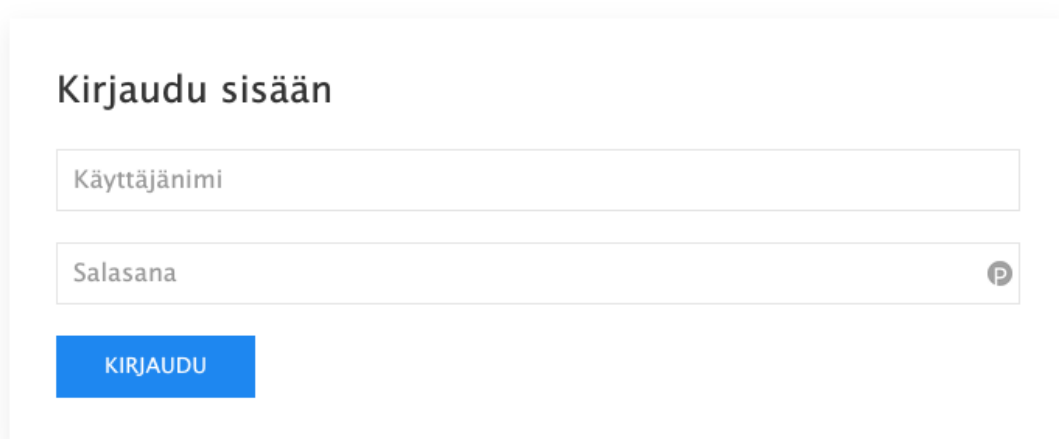
Asiakaspuolen sovelluskehikseksi valitsin Yew'n [25]. Yew muistuttaa hyvin paljon JavaScript-maailmassa suosittua Reactia, eli sen on komponenttipohjainen. Tämä tarkoittaa sitä, että kaikkien ohjelman osien, joilla halutaan näyttää jotakin käyttöliittymän osaa, täytyy implementoida Yew'n Component-rajapintaa. Tässä rajapinnassa on funktiot create, change, update ja view. Näiden funktioiden avulla Yew pystyy orkestroimaan kullakin hetkellä näytettävien komponenttien tilaa.

Tässä projektissa komponentteja oli kolme: ohjelman juuri [liite 3], kirjautumiskomponentti [liite 4] ja suojatun datan noutamiseen tehty komponentti [liite 5].

4.4.3 Ulkonäkö

Olen JavaScript-maailmassa tottunut siihen, että käyttöliittymäkehysiin löytyy usein kirjasto, joka tarjoaa valmiit tyylimääritykset, mutta en löytänyt vastaavaista kirjastoa, joka toimisi Yew'n kanssa. Kaikki käyttöliittymän elementit olivat siis selaimen oletustyyliisiä. En halunnut ryhtyä tässä projektissa kirjoittamaan CSS-määrityksiä alusta alkaen, joten päädyin käyttämään Ulkittiä [26]. Ulkit on CSS-kirjasto, missä on paljon valmiita ja hyvännäköisiä tyylimäärityksiä. Ulkitin dokumentaatio on laaja ja siellä on paljon esimerkkejä, joten alkuun pääsi todella nopeasti [kuva 5].

Login-komponentti löytyy kokonaisuudessaan liitteestä 4. Liitteestä näkee esimerkin Yew'n komponenttirajapinnan käytöstä ja palvelimeen yhdistämisestä. Lisäksi siellä on käytetty Ulkit-tyylityksiä.



Kirjaudu sisään

Käyttäjänimi

Salasana

KIRJAUDU

Kuva 5: Login-komponentin tyylittely, joka on toteutettu käyttämällä UIKit-kirjastoa

4.4.4 Reititys

Reititykseen käytin Yew'n liitännäistä, `yew_routeria`. Reititys asiakaspuolen ohjelmassa tarkoittaa sitä, että selaimen osoitepalkissa olevan polun mukaan käyttäjä reititetään oikeaan ohjelman osaan. Tämä liittyy käsitykseen yhden sivun ohjelmista (katso luku 2.2), joissa selain suorittaa yhden ohjelman, jonka jälkeen perinteisiä sivujen latauksia ei enää tapahdu. Ohjelma muokkaa osoitepalkissa näkyvää osoitetta, jonka pohjalta sitten reititys oikeaan komponenttiin tapahtuu. Käyttöliittymää muokataan käyttämällä DOM:ia, joka mahdollistaa sivun sisällön muokkaamisen ilman sivun uudelleenlataamista.

Asiakaspuolen reititys vaatii palvelinpuolelta sen, että kaikki mahdolliset asiakaspuolen reitit palauttavat ohjelman. Toinen edellytys on se, että palvelin ei palauta uudelleenohjausta (HTTP 302), koska silloin myös osoitepalkissa oleva osoite muuttuu, eikä sitä silloin voida välittää asiakaspuolen ohjelmalle.

Yritin pitkään toteuttaa tällaista logiikkaa palvelinpuolelle siinä onnistumatta, mutta onnekseni `yew_routerin` esimerkeistä löytyi esimerkkikoodia tämän saavuttamiseksi käyttämäni palvelinkehyksen kanssa. Päädyin laittamaan kaikki palvelimen omat reitit polun `/api/` alle, ja asiakaspuolen ohjelman juureen (`/`). Sitten määritin ns. "catch-all" reitin, joka palauttaa asiakaspuolen ohjelman ilman uudelleenohjausta. Tämä on Actixissa nimeltään `default_service`.

Myös yew_routerin kanssa oli omat haasteensa. Alkuun en saanut sitä toimimaan ollenkaan, reititin ohjasi kaikki osoitteet ensimmäisenä määritettyyn reittiin, mikä oli tässä tapauksessa `"/`. Ongelman syyksi paljastui yew_routerin Switch-komponentin kokoava `"to"`-makro (katso luku 3.1.7). Ratkaisuna oli reitin `"/` makron siirtäminen viimeiseksi listassa. Tarkempaa tietoa tästä ongelmasta voi lukea Yew'n dokumentaatiosta [27].

Asiakaspuolen reitityksen toteutuksen voi nähdä liitteessä 3.

4.5 Ongelmat

Projektin aikana ilmeni muutamia ongelmia, joiden takia projekti vaatii vielä jatkokehitystä.

4.5.1 Evästeet

Palvelinpuolella evästeen kokoava kirjasto, Actixin liitännäinen `actix_identity`, kirjoittaa oletuksena evästeeseen parametrin `"HttpOnly": true`, mikä tarkoittaa sitä, että selain ei anna sen sisällä suoritettaville ohjelmille pääsyä tähän evästeeseen. Tämä on hyvä turvallisuusominaisuus, joka estää haitallisia ohjelmia varastamasta käyttäjän kirjautumistietoja.

Suunnittelmana oli käyttää evästeen tietoja asiakaspuolella reititykseen. Esimerkiksi jos evästettä ei ole olemassa, käyttäjä tulee reitittää sisäänkirjautumiskomponenttiin, jossa eväste voidaan noutaa palvelimelta. Koska evästeessä on tämä `HttpOnly`-parametri, tähän ei ole mahdollisuutta. Tämän parametrin voisi tietysti laittaa pois päältä, mutta `actix_identity` ei tarjoa tähän mitään mahdollisuutta.

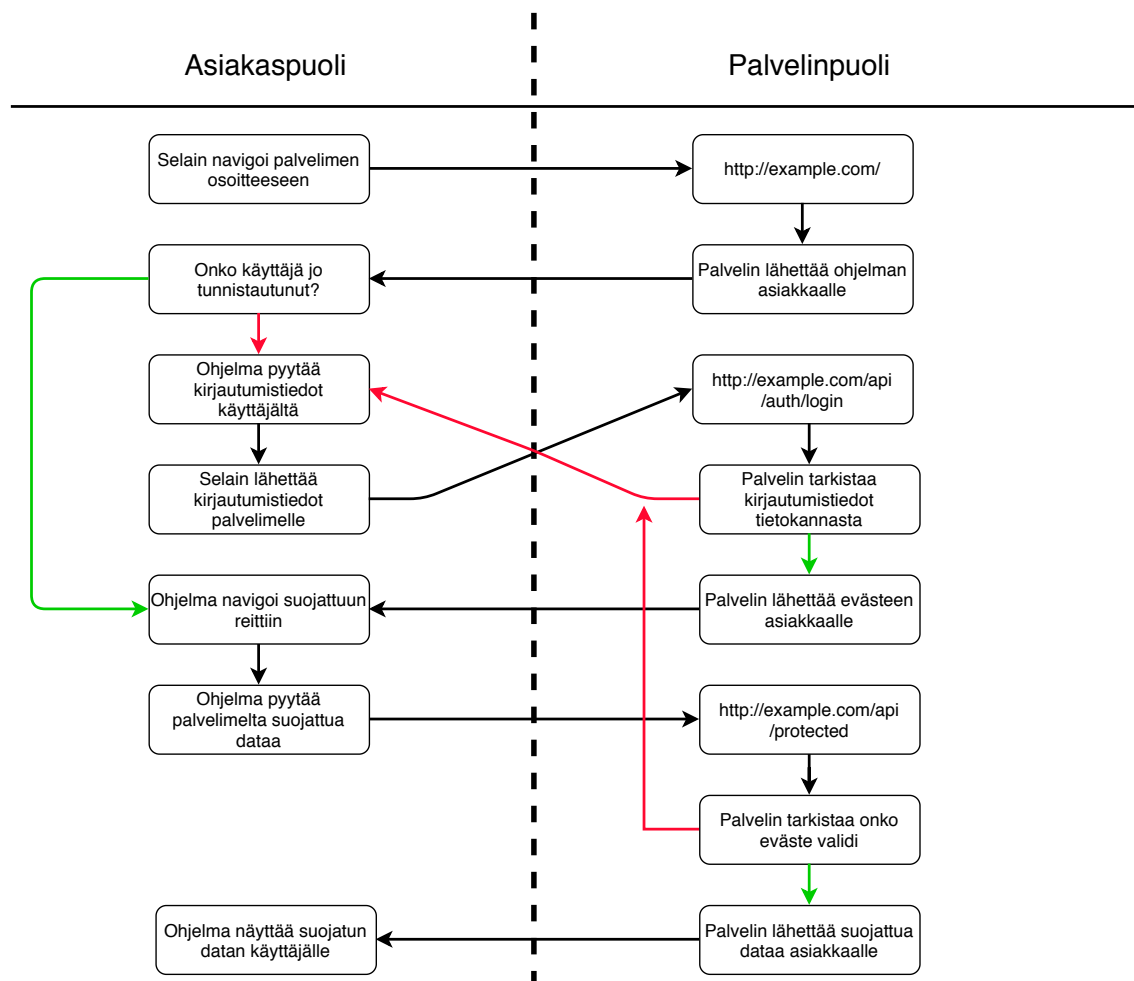
Yhtenä ratkaisuna tähän ongelmaan olisi käyttää JSON Web Tokenin tallettamiseen evästeen sijasta selaimen `LocalStorage`a. Tämä vaatisi lisää työtä sekä asiakaspuolella että palvelinpuolella.

4.5.2 Tietokantayhteys

Palvelimen ja tietokannan välisessä yhteydessä oli koko projektin ajan ongelma, että jokainen kysely tietokantaan kesti noin viisi sekuntia. Luulen että tämä ongelma liittyy jotenkin käyttämäni yhteyksien yhdistämiseen [luku 4.3.4], koska palvelimen loki näyttää satunnaisesti virheviestin Failed to create DB pool: Error(Some("Too many connections")). Asian selvittäminen jää jatkokehitykseen.

4.6 Tulokset

Tuloksena syntyi yksinkertainen web-ohjelma, jossa on toimiva sisäänkirjautumistoiminnallisuus. Sen avulla asiakas voi noutaa palvelimelta suojattua dataa. Kuvassa 6 on määritelty kaikki kirjautumisprosessin vaiheet.



Kuva 6: Kirjautumisprosessi projektissa

Ohjelman juuressa renderöidään komponentti "ProtectedComponent", missä on yksi nappi, jolla voidaan noutaa dataa palvelimelta. Tämä data on suojattu niin, että pyynnön mukana pitää olla eväste, jonka sisällä on JWT, joka sitten validoidaan palvelinpuolella. Jos validointi onnistuu, suojattu data lähetetään asiakkaalle. Asiakaspäässä HTTP-statuskoodin perusteella näytetään joko noudettu data, tai "403 Unauthorized".

Reitissä "/login" renderöidään "LoginComponent", joka koostuu kahdesta tekstikentästä, käyttäjänimi ja salasana, sekä napista, jolla kirjautumistiedot voi lähettää palvelimelle. Onnistumisen kirjautumisen jälkeen palvelimelta lähetetty eväste tallennetaan selaimen muistiin.

Luvussa 4.5.1 mainittujen ongelmien takia toteutus jäi hieman kesken. Eväste saadaan onnistuneesti palvelimelta selaimen, mutta koska eväste ei ole ohjelman saatavilla, en voi toteuttaa tämän perusteella reititystä uudelle sivulle. Tein tämän takia sivun ylälaitaan napit, joilla käyttäjä voi siirtyä manuaalisesti sivulta toiselle. Ajatuksena oli, että onnistuneen kirjautumisen jälkeen käyttäjä siirrettäisiin toiselle sivulle, ja evästeen puuttuessa taas kirjautumissivulle.

5 Rustin soveltuvuus web-ohjelmointiin

5.1 Palvelinpuoli

Palvelinpuolella en näe Rustin käyttämisessä mitään varjopuolia. Jos palvelin tulee osaksi isoja nettisivuja, missä on päivittäin tuhansia kävijöitä, Rustin suorituskyvystä ja turvallisuudesta rinnakkaisajosta voi olla todella paljon hyötyä. Lisäksi jos palvelin tekee joitakin laskennallisesti raskaita tehtäviä, kuten vaikkapa 3D-mallinnusta tai koneoppimista, Rustin suorituskyky tulee entistäkin selvemmäksi.

Näen myös, että Rustin vahvasta tyyppityksestä voi olla paljon hyötyä web-sovelluksen palvelinpuolta rakennettaessa. Kuten koodiesimerkissä 12 nähtiin, vahvan tyyppityksen tuomia etuja voi käyttää hyödykseen rajapintaa rakennettaessa. Lisäksi vahva tyyppitys lisää ohjelman luotettavuutta vähentämällä ajonaikaisten virheiden riskiä. Vahvan tyyppityksen voi kylläkin lisätä myös JavaScriptiin käyttämällä esimerkiksi TypeScriptiä [28].

Lisäksi Rust on suosittu kieli muuallakin kuin web-maailmassa. Tämän ansiosta kirjastojen saatavuus erityisesti palvelinpuolella ei ole ongelma. Rust-koodiin voi myös sisällyttää C- ja C++ -kirjastoja, jos jotakin projektissa tarvittavaa kirjastoa ei löydy Rust-pakettina.

5.2 Asiakaspuoli

Asiakaspuolella Rustin hyötyjä on vaikeampi oikeuttaa. Asiakaspuolen ohjelmat ovat useimmiten yksinkertaisia, koska niiden tehtävänä on usein vain piirtää käyttöliittymä ja välittää dataa palvelimen ja käyttöliittymän välillä. Näin ollen myös suorituskykyvaatimukset ovat todella pienet verrattuna palvelinpuoleen. Lisäksi asiakaspuoli palvelee aina vain yhtä asiakasta kerrallaan verrattuna palvelinpuoleen, jossa usein yksi palvelin palvelee kaikkia asiakkaita. Tästä syystä myös Rustin rinnakkaisajo-ominaisuudet jäävät hyödynämättä.

Vahvan tyyppityksen hyötyjä ei voi kieltää, mutta kuten luvussa 5.1 mainittiin, myös JavaScriptin saa tyyditettyä käyttämällä TypeScriptiä.

Kirjastojen saatavuus on ongelma asiakaspuolella. Kaikkien kirjastojen, jotka vuorovaikuttavat selaimen kanssa, pitää nimenomaan tukea WebAssemblya. WebAssembly-ekosysteemi on vielä melko alkutekijöissään, ja näitä kirjastoja on vielä hyvin vähän. Tämän projektin aikana tätä ei ilmennyt muuta kuin tyylimäärityksissä (luku 4.4.3), mutta tämän projektin käyttöliittymä oli vielä todella yksinkertainen, eikä montaa kirjastoa tarvittukaan.

Jos lähdetään siitä oletuksesta, että suurin osa web-kehittäjistä osaa jo JavaScriptiä, en näiden syiden takia voi suositella kokonaan uuden kielen opettelemista asiakaspuolta varten. Mutta jos on esimerkiksi kehittäjä, jolla ei ole mitään taustaa web-kehityksessä, mutta osaa jo valmiiksi Rustia, tai muita matalan tason kieliä, Rust voi olla mielekäs vaihtoehto. Tässäkin tapauksessa edellytyksenä on, että kaikki kehittäjän tarvitsemat kirjastot ovat saatavilla, tai kehittäjällä on aikaa ja halua toteuttaa puuttuvien kirjastojen toiminallisuus itse.

Rust-koodista käännettyjä WebAssembly-tiedostoja voi myös sisällyttää JavaScript-ohjelmiin [29]. Jos asiakaspuolen ohjelmassa on osia, jotka vaativat suoritustehoa ja/tai

rinnakkaisajo-ominaisuuksia, voi olla hyvä vaihtoehto kirjoittaa vain nämä osat Rustilla ja sisällyttää ne JavaScript-koodiin.

6 Yhteenveto

Rust soveltuu todella hyvin palvelinpuolen web-ohjelmointikieleksi. Sen luotettavuus, suorituskky ja rinnakkaisajo-ominaisuudet tuovat todellisia hyötyjä sekä kehittäjille että palvelimen asiakkaille.

Rust on asiakaspuolen web-ohjelmointikielenä riippuvainen WebAssembly-ekosysteemistä, minkä takia kirjastojen saatavuus on vielä melko huono. Lisäksi asiakaspuolen alhaisten suorituskkyvaatimuksien takia Rustin paremmasta suorituskkyvystä verrattuna JavaScriptiin on vähän hyötyä useimmissa tapauksissa.

Koko projektia ei ole pakko kirjoittaa alusta loppuun Rustilla, niin kuin tässä insinööritöyssä tehtiin. Jos ohjelmassa on korkeampaa suorituskkyä vaativia osia, ne voi kirjoittaa Rustilla ja sisällyttää WebAssembly-binääreinä osaksi muuten JavaScriptillä toteutettua ohjelmaa.

Rustia voi sisällyttää edellä mainituilla tekniikoilla myös olemassa oleviin projekteihin. Jos kehittäjä haluaa siirtää olemassa olevan JavaScript-projektin kokonaan Rust-pohjaiseksi, sen voi tehdä asteittain.

Päätös Rustin käyttämisestä web-ohjelmointikielenä ei ole siis pelkkä kyllä tai ei, vaan se voi olla myös jotakin siltä väliltä. Tältä skaalalta löytyy varmasti jokin ratkaisu, millä jokainen kehittäjä voi hyödyntää Rustia projektissaan omilla ehdoillaan.

Itse tulen todennäköisesti käyttämään Rustia seuraavassa palvelinpuolen projektissani. Tietenkin kehitysympäristö ja tiimitoverien osaaminen vaikuttavat käytettävään kieleen paljon, koska uuden kielen opetteluun kuluu paljon aikaa.

Jään mielenkiinnolla seuraamaan web-kehityksen tulevaisuutta ja sitä, mihin WebAssemblyn ekosysteemi kehittyy lähivuosina. Kirjastojen saatavuuden parantumisen myötä uskon, että se tulee tulevaisuudessa olemaan entistäkin varteenotettavampi vaihtoehto

JavaScriptille. En kuitenkaan usko, että JavaScriptistä pois siirtyminen tapahtuu yhdessä yössä, vaan WebAssembly kehittyy JavaScriptin rinnalla vielä pitkään.

Tämän insinöörityön lähdekoodi on avoin ja se löytyy kokonaisuudessaan GitLabista [30].
Tähän sisältyy projektin lähdekoodi sekä raportin \LaTeX -lähdekoodi.

Lähteet

- 1 Richard Kenneth Eng. The Top 10 Things Wrong with JavaScript. 2016;<<https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8>>. Luettu 5. Maaliskuuta, 2020.
- 2 Wikipedia contributors. Document Object Model. 2020;<https://en.wikipedia.org/wiki/Document_Object_Model>. Luettu 4. toukokuuta 2020.
- 3 Wikipedia contributors. Single-page application. 2020;<https://en.wikipedia.org/wiki/Single-page_application>. Luettu 4. toukokuuta 2020.
- 4 Wikipedia contributors. Representational state transfer. 2020;<https://en.wikipedia.org/wiki/Representational_state_transfer>. Luettu 4. toukokuuta 2020.
- 5 Rust Team. Rust-ohjelmointikielen kotisivu. 2020;<<https://www.rust-lang.org>>. Luettu 24. huhtikuuta, 2020.
- 6 Wikipedian kirjoittajat. Automaattinen roskienkeräys. 2020;<https://fi.wikipedia.org/wiki/Automaattinen_roskienker%C3%A4ys>. Luettu 24. huhtikuuta, 2020.
- 7 Rust Team. What is Ownership - Rust book. 2020;<<https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html>>. Luettu 24. huhtikuuta, 2020.
- 8 Wikipedia contributors. Runtime library. 2020;<https://en.wikipedia.org/wiki/Runtime_library>. Luettu 7. toukokuuta 2020.
- 9 Wikipedia contributors. Thread safety. 2020;<https://en.wikipedia.org/wiki/Thread_safety>. Luettu 7. toukokuuta 2020.
- 10 Aaron Turon. What is Ownership - Rust book. Rust Blog. 2015;<<https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html>>. Luettu 7. toukokuuta, 2020.
- 11 Wikipedia contributors. Understanding WebAssembly text format. 2020;<<https://en.wikipedia.org/wiki/Metaprogramming>>. Luettu 3. toukokuuta 2020.
- 12 Rust Team. Macro std::println. 2020;<<https://doc.rust-lang.org/std/macro.println.html>>. Luettu 3. toukokuuta, 2020.

- 13 Lisp-like DSL for Rust language. 2020. JunSuzukiJapan.
<<https://github.com/JunSuzukiJapan/macro-lisp>>. Luettu 1. toukokuuta 2020.
- 14 The Rust community's crate registry. 2020. Rust Team. <<https://crates.io>>. Luettu 1. toukokuuta 2020.
- 15 WebAssembly homepage. 2020. <<https://webassembly.org>>. Luettu 2. toukokuuta 2020.
- 16 Understanding WebAssembly text format. 2019. MDN contributors. Mozilla. <https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format>. Luettu 2. toukokuuta 2020.
- 17 Wikipedia contributors. S-expression. 2020;<<https://en.wikipedia.org/wiki/S-expression>>. Luettu 7. toukokuuta 2020.
- 18 Rust - Getting started. 2020. Rust Team. <<https://www.rust-lang.org/learn/get-started>>. Luettu 1. toukokuuta 2020.
- 19 Wikipedia contributors. Actor model. 2020;<https://en.wikipedia.org/wiki/Actor_model>. Luettu 16. huhtikuuta, 2020.
- 20 JSON Web Tokens. 2020. Auth0. <<https://jwt.io/introduction>>. Luettu 29. huhtikuuta 2020.
- 21 Wikipedia contributors. Cross-origin resource sharing. 2020;<https://en.wikipedia.org/wiki/Cross-origin_resource_sharing>. Luettu 4. toukokuuta 2020.
- 22 Wikipedia contributors. Same-origin-policy. 2020;<https://en.wikipedia.org/wiki/Same-origin_policy>. Luettu 4. toukokuuta 2020.
- 23 stdweb. 2020. Rust team. <<https://docs.rs/stdweb/0.4.20/stdweb>>. Luettu 1. toukokuuta 2020.
- 24 web_sys. 2020. Rust team. <https://docs.rs/web-sys/0.3.35/i686-unknown-linux-gnu/web_sys>. Luettu 1. toukokuuta 2020.
- 25 Yew Docs. 2020. Yew Team. <<https://yew.rs/docs>>. Luettu 1. toukokuuta 2020.
- 26 UIKit homepage. 2020. <<https://getuikit.com>>. Luettu 2. toukokuuta 2020.
- 27 How to use the Router. 2020. Yew Team. <<https://yew.rs/docs/concepts/router#how-to-use-the-router>>. Luettu 3. toukokuuta 2020.
- 28 TypeScript: Typed JavaScript at Any Scale. 2020. TypeScript team. Microsoft. <<https://www.typescriptlang.org>>. Luettu 3. toukokuuta 2020.
- 29 Using the WebAssembly JavaScript API. 2019. MDN contributors. Mozilla. <<https://developer.mozilla.org/en->

US/docs/WebAssembly/Using_the_JavaScript_API>. Luettu 3. toukokuuta 2020.

- 30 Insinööriyön lähdekoodi. 2020. Marko Korhonen.
<<https://gitlab.com/ReekyMarko/thesis>>. Luettu 8. toukokuuta 2020.

1 Palvelinpuolen käynnistys (main.rs, entrypoint)

```

1  #[macro_use]
2  extern crate diesel;
3  extern crate dotenv;
4
5  pub mod db_connection;
6  pub mod errors;
7  pub mod handlers;
8  pub mod models;
9  pub mod schema;
10 pub mod utils;
11
12 use actix_cors::Cors;
13 use actix_files::{Files, NamedFile};
14 use actix_identity::{CookieIdentityPolicy, IdentityService};
15 use actix_web::{
16     get,
17     http::header,
18     middleware::Logger,
19     web::{delete, get, post, resource, scope},
20     App, Error, HttpResponse, HttpServer,
21 };
22 use chrono::Duration;
23 use db_connection::get_pool;
24 use dotenv::dotenv;
25 use handlers::{authentication, protected};
26
27 pub fn get_env(var_name: &str) -> String {
28     match std::env::var(&var_name) {
29         Ok(var) => return var,
30         Err(e) => {
31             eprintln!(
32                 "Failed to read required environment variable: {}",
33                 &var_name
34             );
35             eprintln!("Reason: {}", e.to_string());
36             panic!("Can't continue without variable");
37         }
38     };
39 }
40
41 async fn serve_index_html() -> Result<NamedFile, Error> {
42     Ok(NamedFile::open("./static/index.html")?)
43 }
44
45 #[get("/api")]

```

```

46 async fn api_404() -> HttpResponse {
47     HttpResponse::NotFound().finish()
48 }
49
50 #[get("/api/{unconfigured_routes:. *}")]
51 async fn api_404_unconfigured() -> HttpResponse {
52     HttpResponse::NotFound().finish()
53 }
54
55 #[rustfmt::skip]
56 #[actix_rt::main]
57 async fn main() -> std::io::Result<()> {
58     std::env::set_var("RUST_LOG", "actix_web=debug,diesel=debug");
59     env_logger::init();
60     dotenv().ok();
61
62     let bind = get_env("BIND");
63     let port = get_env("PORT");
64     let address = format!("{:}:{:}", bind, port);
65
66     println!("Starting server at: http://{:}", &address);
67
68     HttpServer::new(move || {
69         App::new()
70             .wrap(Logger::default())
71             .wrap(
72                 Cors::new()
73                     .allowed_origin(get_env("ALLOWED_ORIGIN").as_str())
74                     .allowed_methods(vec!["GET", "POST", "PUT", "DELETE"])
75                     .allowed_headers(vec![
76                         header::AUTHORIZATION,
77                         header::CONTENT_TYPE,
78                         header::ACCEPT,
79                     ])
80                     .max_age(3600)
81                     .finish(),
82             )
83             .wrap(IdentityService::new(
84                 CookieIdentityPolicy::new(get_env("SECRET").as_bytes())
85                     .domain(get_env("DOMAIN"))
86                     .name(get_env("APP_NAME"))
87                     .path("/")
88                     .max_age(Duration::days(30).num_seconds())
89                     .secure(false),
90             ))
91             .data(get_pool())
92             .service(api_404)
93             .service(
94                 scope("/api/auth")
95                     .service(

```

```

96         resource("/register")
97         .route(post()
98             .to(authentication::register)
99         )
100     )
101     .service(
102         resource("/login")
103         .route(post()
104             .to(authentication::login)
105         )
106     )
107     .service(
108         resource("/logout")
109         .route(post()
110             .to(authentication::logout)
111         ))
112     .service(
113         resource("/delete")
114         .route(delete()
115             .to(authentication::delete)
116         )
117     )
118 )
119 .service(
120     resource("/api/protected")
121     .route(get()
122         .to(protected::protected_route)
123     )
124 )
125 .service(api_404_unconfigured)
126 .service(
127     Files::new("/", "./static")
128     .index_file("index.html")
129 )
130 .default_service(get().to(serve_index_html))
131 })
132 .bind(address)?
133 .run()
134 .await
135 }
```


2 JWT-tunnisteen luominen ja vahvistaminen

```

1  extern crate bcrypt;
2  extern crate jsonwebtoken;
3
4  use crate::get_env;
5  use actix_web::HttpResponse;
6  use chrono::{Duration, Local};
7  use jsonwebtoken::{decode, encode, DecodingKey, EncodingKey, Header,
    Validation};
8  use serde::{Deserialize, Serialize};
9
10 #[derive(Debug, Serialize, Deserialize)]
11 struct Claims {
12     sub: i32,
13     name: String,
14     admin: bool,
15     exp: usize,
16 }
17
18 #[derive(Deserialize)]
19 pub struct UserWithToken {
20     pub id: i32,
21     pub username: String,
22     pub admin: bool,
23 }
24
25 impl From<Claims> for UserWithToken {
26     fn from(claims: Claims) -> Self {
27         UserWithToken {
28             id: claims.sub,
29             username: claims.name,
30             admin: claims.admin,
31         }
32     }
33 }
34
35 impl Claims {
36     fn with_username(id: i32, username: &str, admin: bool) -> Self {
37         Claims {
38             sub: id,
39             name: username.into(),
40             admin,
41             exp: (Local::now() + Duration::hours(24)).timestamp() as
    usize,
42         }
43     }

```

```

44 }
45
46 pub fn encode_token(id: i32, username: &str, admin: bool) ->
    Result<String, HttpResponse> {
47     let claims = Claims::with_username(id, username, admin);
48     let secret = get_env("JWT_SECRET");
49     let key = EncodingKey::from_secret(secret.as_bytes());
50
51     encode(&Header::default(), &claims, &key)
52         .map_err(|e|
53             HttpResponse::InternalServerError().json(e.to_string()))
54 }
55
56 pub fn decode_token(token: &str) -> Result<UserWithToken, HttpResponse> {
57     let secret = get_env("JWT_SECRET");
58     let key = DecodingKey::from_secret(secret.as_bytes());
59
60     decode::<Claims>(token, &key, &Validation::default())
61         .map(|data| data.claims.into())
62         .map_err(|e| HttpResponse::Unauthorized().json(e.to_string()))
63 }

```

3 Asiakaspuolen käynnistys (main.rs, entrypoint)

```

1  #![recursion_limit = "256"]
2
3  extern crate log;
4  extern crate web_logger;
5
6  mod component;
7  pub mod utils;
8
9  use component::{login::LoginComponent, protected::ProtectedComponent};
10 use yew::prelude::*;
11 use yew::virtual_dom::VNode;
12 use yew_router::{prelude::*, switch::Permissive, Switch};
13
14 struct App {}
15
16 #[derive(Debug, Switch, Clone)]
17 enum AppRoute {
18     #[to = "/login"]
19     Login,
20     #[to = "/"
21     Root,
22     PageNotFound(Permissive<String>),
23 }
24
25 impl Component for App {
26     type Message = ();
27     type Properties = ();
28
29     fn create(_: Self::Properties, _link: ComponentLink<Self>) -> Self {
30         App {}
31     }
32
33     fn update(&mut self, _msg: Self::Message) -> ShouldRender {
34         true
35     }
36
37     fn view(&self) -> VNode {
38         html! {
39             <div>
40                 <nav class="menu",>
41                     <RouterButton<AppRoute> route=AppRoute::Login>
42                         {"Kirjautuminen"} </RouterButton<AppRoute>>
43                     <RouterButton<AppRoute> route=AppRoute::Root>
44                         {"Suojattu data"} </RouterButton<AppRoute>>
45                 </nav>

```

```

44         <Router<AppRoute>
45             render = Router::render(|switch: AppRoute| {
46                 match switch {
47                     AppRoute::Login => html!{<LoginComponent />},
48                     AppRoute::PageNotFound(Permissive(None)) =>
html!{"Page not found"},
49
AppRoute::PageNotFound(Permissive(Some(missed_route))) =>
html!{format!("Page '{}' not found", missed_route)},
50                     AppRoute::Root => {
51                         html!{<ProtectedComponent />}
52                     },
53                 }
54             })
55             redirect = Router::redirect(|route: Route| {
56
AppRoute::PageNotFound(Permissive(Some(route.route)))
57                 })
58             />
59         </div>
60     }
61 }
62 }
63
64 fn main() {
65     web_logger::init();
66     yew::start_app::<App>();
67 }

```

4 Sisäänkirjautuminen

```

1 use log::{error, info};
2 use serde_json::json;
3 use yew::format::Json;
4 use yew::prelude::*;
5 use yew::services::fetch::{FetchService, FetchTask, Request, Response};
6
7 pub struct LoginComponent {
8     component_link: ComponentLink<LoginComponent>,
9     username: String,
10    password: String,
11    login_button_disabled: bool,
12    fetch_service: FetchService,
13    fetch_task: Option<FetchTask>,
14    fetching: bool,
15 }
16
17 pub enum Msg {
18     UpdateUsername(String),
19     UpdatePassword(String),
20     HandleForm(),
21     FetchReady(String),
22     FetchError,
23 }
24
25 impl LoginComponent {
26     fn update_button_state(&mut self) {
27         self.login_button_disabled = self.username.is_empty() ||
28         self.password.is_empty();
29     }
30
31     fn post_login(&mut self) {
32         self.fetching = true;
33
34         let user = json!({
35             "username": self.username,
36             "password": self.password
37         });
38
39         let request = Request::builder()
40             .method("POST")
41             .header("Content-Type", "application/json")
42             .uri("http://localhost:3880/api/auth/login")
43             .body(Json(&user))
44             .unwrap();

```

```
45         let callback =
46             self.component_link
47                 .callback(|response: Response<Result<String,
anyhow::Error>>| {
48                 let (meta, body) = response.into_parts();
49                 if meta.status.is_success() {
50                     Msg::FetchReady(body.unwrap())
51                 } else {
52                     Msg::FetchError
53                 }
54             });
55
56         let task = self.fetch_service.fetch(request, callback);
57         self.fetch_task = Some(task.unwrap());
58     }
59 }
60
61 impl Component for LoginComponent {
62     type Message = Msg;
63     type Properties = ();
64
65     fn create(_: Self::Properties, link: ComponentLink<Self>) -> Self {
66         Self {
67             component_link: link,
68             username: String::new(),
69             password: String::new(),
70             login_button_disabled: true,
71             fetch_service: FetchService::new(),
72             fetch_task: None,
73             fetching: false,
74         }
75     }
76
77     fn change(&mut self, _: Self::Properties) -> ShouldRender {
78         true
79     }
80
81     fn update(&mut self, msg: Self::Message) -> ShouldRender {
82         match msg {
83             Msg::UpdateUsername(new_username) => {
84                 self.username = new_username;
85                 self.update_button_state();
86             }
87
88             Msg::UpdatePassword(new_password) => {
89                 self.password = new_password;
90                 self.update_button_state();
91             }
92
93             Msg::HandleForm() => self.post_login(),
```

```

94
95         Msg::FetchReady(response) => {
96             self.fetching = false;
97             info!("Login successful: {}", response);
98         }
99
100        Msg::FetchError => {
101            self.fetching = false;
102            error!("There was an error connecting to API")
103        }
104    }
105    true
106 }
107
108 fn view(&self) -> Html {
109     let onclick = self.component_link.callback(|_| Msg::HandleForm());
110     let oninput_username = self
111         .component_link
112         .callback(|e: InputData| Msg::UpdateUsername(e.value));
113     let oninput_password = self
114         .component_link
115         .callback(|e: InputData| Msg::UpdatePassword(e.value));
116
117     html! {
118         <div class="uk-card uk-card-default uk-card-body
119         uk-width-1-3@s uk-position-center">
120             <h1 class="uk-card-title">{ "Kirjaudu sisään" }</h1>
121             <div>
122                 <fieldset class="uk-fieldset">
123                     <input class="uk-input uk-margin",
124                         placeholder="Käyttäjänimi",
125                         disabled=self.fetching,
126                         value=&self.username,
127                         oninput=oninput_username, />
128                     <input class="uk-input uk-margin-bottom",
129                         placeholder="Salasana",
130                         disabled=self.fetching,
131                         type="password",
132                         value=&self.password,
133                         oninput=oninput_password, />
134                     <button
135                         class="uk-button uk-button-primary",
136                         type="button",
137                         disabled=self.fetching,
138                         onclick=onclick>
139                         { "Kirjaudu" }
140                     </button>
141                 </fieldset>
142             </div>
143         </div>

```

143 }
144 }
145 }

5 Suojatun tiedon noutaminen

```

1 use log::{error, info};
2 use yew::format::Nothing;
3 use yew::prelude::*;
4 use yew::services::fetch::{FetchService, FetchTask, Request, Response};
5
6 pub struct ProtectedComponent {
7     component_link: ComponentLink<ProtectedComponent>,
8     fetch_service: FetchService,
9     fetch_task: Option<FetchTask>,
10    fetching: bool,
11    data: String,
12 }
13
14 pub enum Msg {
15     FetchData(),
16     FetchReady(String),
17     FetchError,
18 }
19
20 impl ProtectedComponent {
21     fn get_data(&mut self) {
22         self.fetching = true;
23
24         let request = Request::get("http://localhost:3880/api/protected")
25             .body(Nothing)
26             .unwrap();
27
28         info!("Request: {:?}", request);
29
30         let callback =
31             self.component_link
32                 .callback(|response: Response<Result<String,
33 anyhow::Error>>| {
34
35                     let (meta, body) = response.into_parts();
36                     info!("{}", meta.status);
37                     if meta.status.is_success() {
38                         Msg::FetchReady(body.unwrap())
39                     } else {
40                         Msg::FetchError
41                     }
42                 });
43
44         let task = self.fetch_service.fetch(request, callback);
45         self.fetch_task = Some(task.unwrap());
46     }
47 }

```

```

45 }
46
47 impl Component for ProtectedComponent {
48     type Message = Msg;
49     type Properties = ();
50
51     fn create(_: Self::Properties, link: ComponentLink<Self>) -> Self {
52         Self {
53             component_link: link,
54             fetch_service: FetchService::new(),
55             fetch_task: None,
56             fetching: false,
57             data: String::from(""),
58         }
59     }
60
61     fn change(&mut self, _: Self::Properties) -> ShouldRender {
62         true
63     }
64
65     fn update(&mut self, msg: Self::Message) -> ShouldRender {
66         match msg {
67             Msg::FetchReady(response) => {
68                 self.fetching = false;
69                 info!("Fetch successful: {}", response);
70                 self.data = response;
71             }
72
73             Msg::FetchError => {
74                 self.fetching = false;
75                 error!("There was an error connecting to API");
76                 self.data = String::from("401 Unauthorized");
77             }
78
79             Msg::FetchData() => {
80                 self.fetching = true;
81                 self.get_data();
82             }
83         }
84         true
85     }
86
87     fn view(&self) -> Html {
88         let onclick = self.component_link.callback(|_| Msg::FetchData());
89
90         html! {
91             <div class="uk-card uk-card-default uk-card-body
92 uk-width-1-3@s uk-position-center">
93                 <h1 class="uk-card-title">{ "Suojattu data" }</h1>
94                 <p>{&self.data}</p>

```

```
94         <button
95         class="uk-button uk-button-primary",
96         type="button",
97         disabled=self.fetching,
98         onclick=onclick>
99             { "Hae data" }
100         </button>
101     </div>
102 }
103 }
104 }
```

6 Tietokannan migraatitiedostot

6.1 up.sql

```
1 CREATE TABLE users (  
2   `id` int NOT NULL AUTO_INCREMENT,  
3   `username` varchar(100) UNIQUE NOT NULL,  
4   `password` varchar(128) NOT NULL,  
5   `admin` boolean NOT NULL,  
6   `created_at` timestamp NOT NULL,  
7   PRIMARY KEY (`id`)  
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

6.2 down.sql

```
1 DROP TABLE users;
```