Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Arto Jussilainen

Toimittajarahoitus-web-sovelluksen suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

12.4.2019

Tekijä Otsikko Sivumäärä Aika	Arto Jussilainen Toimittajarahoitus-web-sovelluksen suunnittelu ja toteutus 53 sivua 12.4.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander CTO Antti Marjala

Tämän insinöörityön aiheena oli tehdä Apix Messaging Oy:lle uusi myytävä sovellus Toimittajarahoitus. Toimittajarahoitus on selaimessa käytettävä sovellus, jossa yritysten rahoitustarpeet täyttyvät. Laskuntoimittaja eli myyjä voi hakea rahoitusta järjestelmään verkkolaskuina siirtämillensä laskuille ja hyväksymisprosessin jälkeen saada saatavat, joista osa menee korkotuloina rahoittajalle. Ostaja hyötyy pidennetystä maksuajasta, jonka aikana hän voi käyttää varojansa muihin tarkoituksiin ja saada saatavia tuottamistansa palveluista. Rahoittaja ottaa riskin maksamattomista laskuista.

Toimittajarahoitus sisältää toimittajien automaattisen kutsumisen ja rekisteröinnin käyttämään sovellusta, ostajan hallitun ostolaskujen hyväksymisen rahoitukseen ja rahoittajan täyden kontrollin rahoitukseen siirrettävistä saatavista. Järjestelmän pohjana käytetään muokattua myyntilaskujenrahoitusjärjestelmää ja toteuttavaksi insinöörityölle jää käyttöliittymä sekä rajapinta, joka kommunikoi edellä mainitun järjestelmän kanssa.

Toimittajarahoituksen käyttöliittymä toteutettiin Vue.js JavaScript -kehyksellä. Käyttöliittymä sisältää välttämättömät näkymät ja toiminnot käyttäjä- ja laskunhallinnan toteuttamiseen. Käyttäjät pääsevät kutsulla järjestelmään, jonka hallitsijana toimii rahoittaja. Käyttäjän oikeudet määräytyvät tämän suhteesta järjestelmään, mikä heijastuu myös käyttöliittymässä. REST-rajapinta toteutettiin Googlen kehittämällä ohjelmointikielellä Golangilla. Käyttöliittymä kommunikoi rajapinnan kanssa, joka vuorostaan kommunikoi MySQL-relaatiotietokannan ja verkkolaskujärjestelmän kanssa.

Insinöörityön lopputuloksena saatiin uusi myytävä tuote sekä komponentteja ja koodipohjaa hyödynnettäviksi muihin projekteihin. Työn kautta nousi myös uusia ideoita ja tapoja asioiden toteuttamiselle niin Vuella kuin Golangillakin. Sovellus jää jatkuvan kehityksen alaiseksi, jolloin sitä parannellaan ja siihen lisätään ominaisuuksia tarpeiden mukaan.

Avainsanat	Vue.js, Golang, rahoitus

Author Title Number of Pages Date	Arto Jussilainen Seller Financing Web Application 53 pages 12 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Antti Marjala, Chief Technical Officer

The purpose of this thesis was to create a new sellable product called Seller Financing for Apix Messaging Oy. Seller Financing is a web application, where companies' financing needs are met. The seller can transfer invoices to the platform through e-invoicing and after being accepted for financing receive funds from which the financer takes an interest fee. In turn, the buyer will benefit from a longer payment period in which they can make other investments as well as receive payments from their own services. The financer assumes the risk of unpaid bills.

Seller Financing includes the ability for automated seller invitation and registration to the system, buyers' control of approving purchase invoices to be financed and financers' full control of funds. Seller Financing's message processing system is based on a modified sales invoice financing system. This leaves creating a user interface and API, which communicates with the aforementioned messaging system, for the project.

The user interface for Seller Financing was built with Vue.js JavaScript framework and it includes all the necessary functionality for managing invoices and users. Seller Financing is an invitation only based system and controlled by the financer. The rights of the user are based on their role in the system, which is also reflected on the user interface. REST-API for the application was built with the programming language from Google, Golang. The user interface communicates with the API, which in turn communicates with a MySQL relational database and the messaging system.

As the result of the study, a new product was built, along with new components and reusable code assets for other projects. New ideas and practices for building applications with Vue and Golang were also gathered during the process. The application is left under continuous development, where it will be further improved and expanded with new features according to specific needs.

Keywords	Vue.js, Golang, financing

Sisällys

Lyhenteet

ORM	Object-relational mapping. Esitetään oliomallin mukainen kuvaus relaatiomallin mukaisena.
DOM	Document Object Model. Kuvaa HTML:n rakennetta puuna.
HTML	Hypertext Markup Language. Merkkauskieli verkkosivujen kuvaamiseen.
HTTP	Hypertext Transfer Protocol. Protokolla tiedon siirtämiseen selainten ja palvelimien välillä.
JSON	JavaScript Object Notation. Tiedostomuoto, joka esittää tiedon avain-arvo-pareina.

1 Johdanto

Tarvitseeko yrityksesi lisää maksuaikaa laskuille tai haluaako se laskusaatavat välittömästi? Haluatko tukea yritysten toimintaa rahoittajana? Toimittajarahoitus on Apix Messaging Oy:n uusi myytävä sovellus, jossa yritysten rahoitustarpeet täyttyvät. Ostaja voi kutsua laskuntoimittajansa alustalle ja vastaanottaa laskunsa rahoittamista varten, jolloin rahoittaja myöntää hänelle lisää maksuaikaa. Toimittaja voi lähettää laskunsa verkkolaskuina järjestelmään ja rahoittamisen myötä saada saatavat välittömästi, sen sijaan että joutuisi odottamaan pitkän maksuajan. Rahoittaja hallitsee koko tapahtumaketjua ja tienaa korkosaatavia rahoitetuista laskuista.

Apix Messaging Oy on vuonna 2010 perustettu yritys, jossa henkilöidensä kautta tiivistyy vahva kokemus yritysten liiketoimintatransaktioiden välityksestä, kuten verkkolaskutuksesta. Toimittajarahoitus on Apixille uusi lisäys verkkolaskutusta hyödyntävien sovellusten kokoelmaan.

Toimittajarahoituksen tarkoituksena on palvella rahoitustoiminnan kaikkia osapuolia nopeasti ja vaivattomasti. Siihen sisältyy osittain automatisoitu toimittajien kutsuminen ja rekisteröinti käyttämään alustaa, ostajan hallittu ostolaskujen hyväksyminen rahoitukseen ja rahoittajan täysi kontrolli rahoitukseen siirrettävistä saatavista.

Toimittajarahoitussovellus muodostuu kahdesta eri osasta: käyttöliittymästä ja varsinaisesta aineiston käsittelyjärjestelmästä. Käyttöliittymä ja siihen liittyvät rekisteröinti sekä laskun hyväksymisprosessi ovat täysin uusia. Aineiston käsittelyjärjestelmän pohjana käytetään aikaisempaa myyntilaskujen rahoitusjärjestelmää, jota muokataan tarjoamaan riittävät palvelut uudelle käyttöliittymälle.

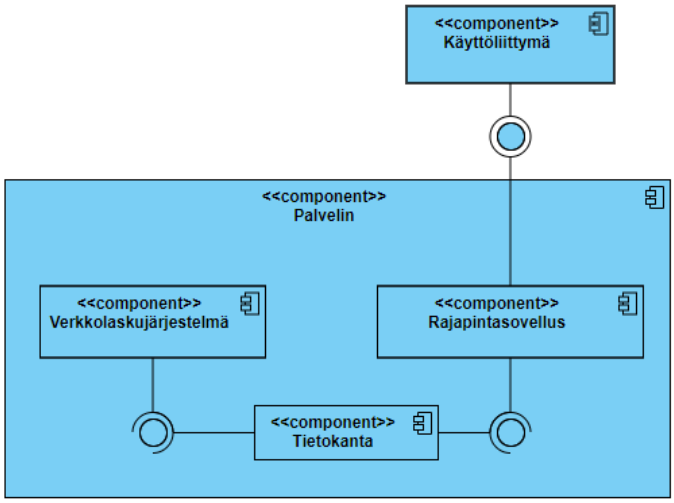
Insinöörityöllä oli tavoitteena toteuttaa moderneilla teknologiolla tämä palvelu siten, että käyttöliittymänä toimii selainpohjainen ratkaisu ja palvelimelle toteutetaan rajapintasovellus, joka on yhteydessä käyttöliittymään ja käsittelyjärjestelmään.

Tässä raportissa käyn läpi toimittajarahoituksen suunnittelua, käytössä olleita teknologioita, työn toteutusta ja lopputulosta.

2 Suunnitelma ja määrittely

Rahoitustoiminta hyödyntää kaikkia osapuolia. Yritykset pyrkivät saamaan maksun teettämistänsä palveluista heti ja samaan aikaan viivyttää omien hankintojensa maksamista, sillä kumpikin asia vaikuttaa käyttöpääomaan pienentävästi. Käyttöpääoma tarkoittaa juoksevaan liiketoimintaan sidottua omaisuutta, mitä pienemmällä määrällä yritys pärjää, sitä parempi [1]. Rahoitustoiminnassa myyjä saa saatavansa heti myydessään laskunsa rahoittajalle ja tällöin maksamattoman laskun riski siirtyy rahoittajalle. Rahoittaja hyötyy ottamalla myyjälle tehdystä maksusta korko-osuuden. Laskun maksaja, eli ostaja, saa lisää maksuaikaa, jonka aikana hän voi tehdä esimerkiksi hankintoja ja saada saatavia omista palveluistansa [2]. Tämän toiminnan toteuttamiseksi toimittajakeskeisestä näkökulmasta lähdettiin liikkeelle Toimittajarahoitussovelluksessa.

Toimittajarahoitussovelluksen avaintoiminnallisuudet ovat toimittajien osittain automatisoitu kutsuminen ja rekisteröityminen järjestelmään, käyttäjien hallinta, laskujen siirtyminen sisään ja pois järjestelmästä, laskujen hyväksyminen rahoitusta varten sekä laskun tilanteen seuranta. Näiden toiminnallisuuksien toteuttamiseen tarvitaan käyttöliittymä ja rajapinta, jonka kanssa se kommunikoi. Käyttöliittymäksi tulee selainpohjainen sovellus ja rajapinnaksi palvelimella pyörivä sovellus, joka on yhteydessä tietokantaan (kuva 1). Palvelimelle tulee myös verkkolaskujärjestelmä, jonka tehtävänä on lukea verkkolaskuja tietokantaan ja muodostaa verkkolaskuja lähetettäväksi eteenpäin.



Kuva 1. Toimittajarahoituspalvelun komponentit

Toimittajarahoituksen verkkolaskujärjestelmä luodaan aikaisemman myyntilaskujen rahoitusjärjestelmän pohjalta pienin muokkauksin. Myös tietokanta on sitä myöten suurimmilta osin rakennettu valmiiksi. Työksi jää käyttöliittymä ja rajapinta.

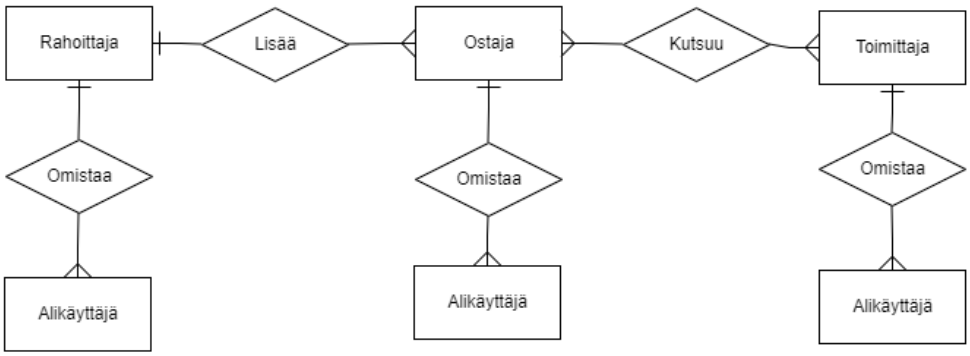
Käyttöliittymän suunnittelussa keskityttiin muutamaan avainnäkymään ja suunnittelussa otettiin mallia aikaisemmista laskunkäsittelyyn erikoistuvista sovelluksista. Sovelluksen ollessa sisäisessä kehityksessä oli helppo reagoida muutoksiin, joita tulikin matkan varrella. Toimittajarahoituksen käyttöliittymässä keskityttiin loogisiin kokonaisuuksiin, helppokäyttöisyyteen, nopeuteen ja selkeään ulkoasuun. Rajapinnan kehys suunniteltiin aikaisempien palvelinpään sovellusten pohjalta ja mukautettiin uuteen tietokantaan.

Sisällöltään Toimittajarahoitus pyörii laskujen ympärillä. Laskut kuitenkin liikkuvat ihmisten ja yritysten välillä ja rahoituslaskuihin liittyvät toimittajat, ostajat ja rahoittajat, joten toiseksi tärkeäksi teemaksi nousevat käyttäjät ja näiden hallinta.

2.1 Käyttäjät

Toimittajarahoituksessa käyttäjät ovat keskeisessä asemassa. Kukin toimii yleensä yrittäjän tai toiminimen alaisena y-tunnukseen yhdistettynä. Käyttäjät voidaan jakaa kolmeen eri rooliin: rahoittajaan, joka on myös järjestelmän admin, ostajaan ja toimittajaan. Rooleilla on selkeä hierarkia, rahoittajalla on täydet oikeudet järjestelmään, tämä lisää ostajat, jotka puolestaan kutsuvat toimittajia. Toimittaja hyväksyy kutsun ja antaa rekisteröitymistiedot, jonka jälkeen rahoittaja hyväksyy toimittajan järjestelmään. Hierarkia on kuvattu kuvassa 2. Kaikilla rooleilla on myös pää- ja alitason käyttäjiä. Pääkäyttäjällä on roolille kuuluvat täydet oikeudet, kun taas alikäyttäjällä on rajoitetut oikeudet. Pääkäyttäjä vastaa alikäyttäjien luomisesta ja hallinnoimisesta. Alikäyttäjien tarkoitus on lähinnä avustaa manuaalisessa työssä eli laskujen käsittelemisessä.

Toimittaja-nimitys viittaa laskujen toimitukseen, eli käytännössä toimittaja on myyjä. Ostaja taas on yksiselitteinen, ja viittaa tahoon, joka vastaanottaa laskun. Rahoittaja puolestaan rahoittaa valitsemansa laskut, joille on pyydetty rahoitusta.



Kuva 2. Havainnollistus käyttäjähierarkiasta

Tietokantatasolla käyttäjät koostuvat yritys- ja käyttäjärelaatioista. Yrityksillä kuten käyttäjillä on omistajuussuhde toisiin relaatioihin nähden. Yritys ja käyttäjärelaatiot koostuvat perustiedoista.

Käyttäjähallinnan tärkeyden vuoksi sille kuuluu monta näkymää käyttöliittymästä. Näkymien määrä johtuu myös eri käyttäjäryhmien eri tarpeista. Näkymiin kuuluu käyttäjät-osio, joka kuuluu kaikille pääkäyttäjille käyttäjien hallitsemista varten, sekä toimittajat- ja pohjat-osiot, jotka kuuluvat ostajille. Toimittajat- ja pohjat-näkymät palvelevat automatisoitua toimittajan kutsumista, jossa toimittajat-näkymässä täytetään toimittajien tietoja ja pohjat-näkymässä tehdään kutsuja toimittajien kutsumista varten. Aktivointi-näkymä on sivu, jonne toimittaja saapuu vastaanottaessaan ostajan kutsun ja jossa hänestä on esitäytettyjä tietoja, joita hän voi korjata rekisteröitymistä varten.

2.2 Laskut

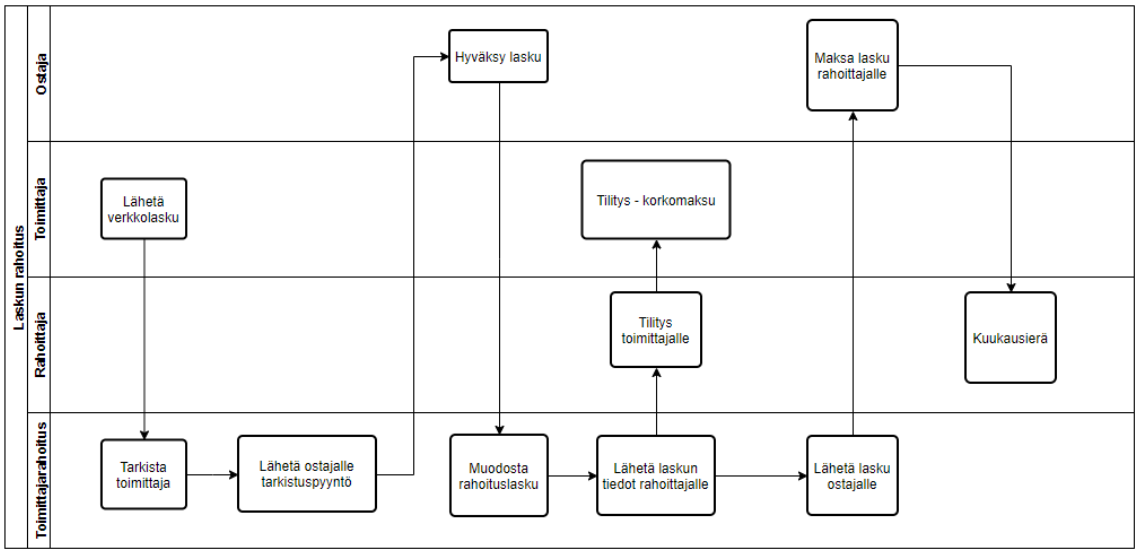
Verkkolaskujen välittämiseen liittyvä infrastruktuuri löytyy jo, ja toimittajarahoitukselle jää tehtäväksi mahdollistaa laskuihin kohdistuvaa päätöksentekoa. Käyttöliittymän yhdessä rajapinnan kanssa tulee tarjota tämä käyttäjille.

Toimittajarahoituksen varsinainen sisältö ovat laskut. Laskut ovat strukturoitua tietoa ostajasta, myyjästä ja tuotteista, mikä istuu hyvin relaatiotietokantaan. Käyttöliittymässä

laskut sopivat esitettäviksi taulukossa, johon mahtuu tiheästi paljon tietoja, eli tässä tapauksessa jokaista riviä vastaa yksi lasku. Taulukon tulee tarjota mahdollisuus suodattaa laskuja tekstihaun ja tilan mukaan sekä selata ja järjestellä laskuja. Laskuihin kuitenkin liittyvät myös kuvat ja liitteet, joita ei pysty esittämään taulukkonäkymässä, joten yksittäisen laskun tarkastelemiseen tarvitaan myös yksittäisnäkymä, josta voi myös ladata kuvat ja liitteet. Laskuja tulee myös pystyä käsittelemään ja kommentoimaan käsittelyn yhteydessä. Tätä varten tulee olla palkki toimintoja varten.

Laskujen käsittelyn tulee olla helppoa ja nopeaa. Rivejä pitää olla helppo valita: yksi tai monta kerrallaan, jonka jälkeen painikkeiden avulla laskujen tilaa muutetaan siten, että laskunkäsittelyjärjestelmä ottaa ne prosessiin. Laskut siirtyvät sisään ja ulos järjestelmästä verkkolaskuina, jotka sisältävät XML-tiedoston ja mahdollisia kuvatiedostoja. Toimittaja lähettää laskut järjestelmään kohdistamalla ne verkkolaskuosoitteeseen. Verkkolaskut luetaan tietokantaan prosessin kautta, jolloin niitä voidaan hakea käyttöliittymään näytettäväksi rajapinnan kautta.

Laskunkäsittelyprosessissa lasku tulee käyttöliittymään ensimmäiseksi toimittajalle, joka voi pyytää rahoitusta laskulle tai poistaa sen. Rahoituspyynnön jälkeen ostaja voi vahvistaa laskun näkymässään hyväksymällä sen, jolloin lasku siirtyy rahoitettavaksi. Myös rahoittaja voi hyväksyä suoraan laskun ilman ostajan hyväksyntää, jos tämä haluaa ottaa riskin. Todellisuudessa ostajan hyväksyessä laskun hän vahvistaa, että lasku oikeasti kuuluu hänelle. Ostajan hylätessä laskun voi rahoittaja palauttaa sen uudestaan käsittelyyn, kuten myös virhetilanteissa. Rahoittaja hylkää laskut poistamalla ne. Laskun käsittelemisen voi pysäyttää asettamalla sen pitoon ostajan tai rahoittajan toimesta ja voi palauttaa taas käsiteltäväksi vapauttamalla sen. Kaikissa vaiheissa laskuun voi asettaa kommentin. Kun rahoittaja on hyväksynyt laskun, siitä muodostetaan rahoituslasku, joka lähetetään ostajalle, ja rahoittaja maksaa toimittajalle. Rahoitusprosessi on kuvattu kuvassa 3.



Kuva 3. Laskun rahoitus

2.3 Teknologiat

Toimittajarahoituksen toteutukseen käytettävä teknologia valittiin käytettävyyden, tuen ja kokemusten kannalta. Käytettävyys tarkoittaa tässä tapauksessa sitä, kuinka iso oppimiskynnys teknologialla on ja kuinka helppoa sillä on toteuttaa sovellus. Teknologian tuki ilmenee dokumentoinnissa ja siinä, onko teknologia avoimen lähdekoodin piirissä. Teknologian elinkaarta voi myös yrittää arvioida, joskin se on vaikeaa. Käyttäjämäärien avulla voi kuitenkin arvioida, onko teknologialla kullakin hetkellä yhteisöä ympärillänsä ja kehitetäänkö sitä enää, vai onko se kuolemassa. Yhteisön koko vaikuttaa myös teknologialle tuotettujen valmiiden komponenttien ja kirjastojen määrään positiivisesti. Teknologioiksi valittiin käyttöliittymän toteuttamiseen Vue.js ja palvelinpuolen REST-rajapinta toteutukseen Golang. Kokemusta kummankin teknologian käytöstä löytyi talon sisältä. Tietokantaratkaisuna pysyy jo paljon käytetty relaatiotietokanta MySQL.

Käyttöliittymän vaihtoehtoina olivat React.js ja Vue.js, kummassakin löytyy hyvä dokumentaatio ja tuki, vaikkakin Reactia käytetään paljon enemmän [3]. Käyttäjämäärien eron voi todeta myös simppelillä Stackoverflow-haulla. Hakusanalla ”vue” löytyi 44 211 tulosta, kun taas ”react” tuotti 257 935 tulosta (1.3.2019). Stackoverflow-kysymysten

määrä tarkoittaa myös sitä, että Reactin kanssa kohdattavaan ongelmaan olisi todennäköisempää löytää ratkaisu. Vue oli jo kuitenkin käytössä, joten sillä oli luonnollista jatkaa. Vuen käyttökokemus koettiin myös hyväksi ja oppimiskynnys matalaksi. Käyttökokemusta tukee myös komentorivityökalu, Vue CLI, jolla voi muun muassa luoda projektille pohjan.

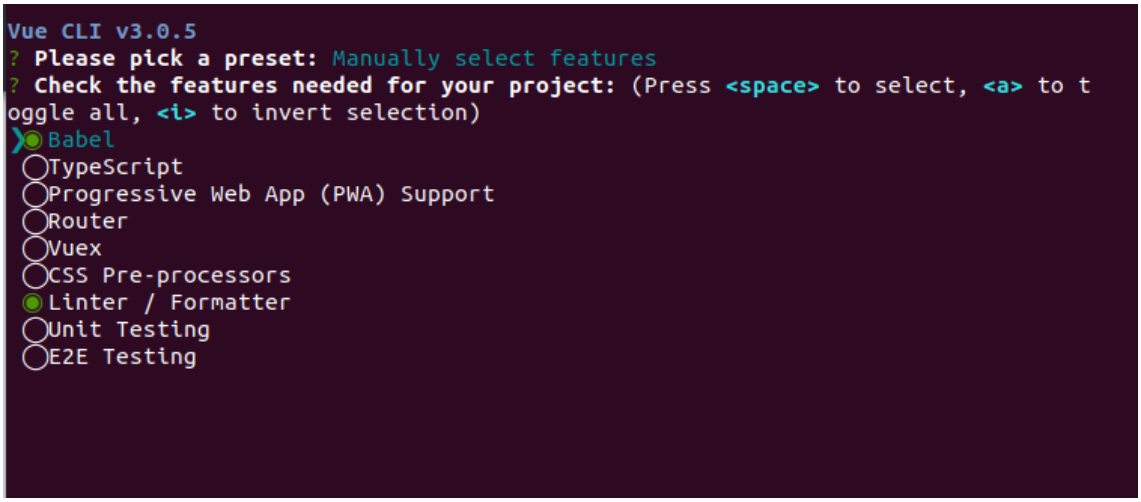
Golangin valintaan vaikutti vahvasti aikaisempi käyttö. Kyseisellä kielellä löytyi jo toteutettuja palvelinsovelluksia, joten käyttökokemusta ja koodipohjaa oli jo valmiina. Golangin dokumentaatio on hyvää ja yhteisön tuottamia kirjastoja löytyy rutkasti.

3 Vue.js

Vue.js on JavaScript-kehys, joka Reactin ja Emberin tavoin hyödyntää virtuaalista DOMia (Document Object Model). DOM kuvaa verkkosivun HTML-rakennetta puun muodossa ja siihen liittyvän rajapinnan kautta voi luoda ja muokata HTML-elementtejä. Virtuaaliseen DOMiin voi kohdistaa muutoksia samalla tavalla, jotka vasta sen jälkeen kohdistetaan oikeaan DOMiin hallitusti. Tämä tekee HTML-elementtien päivittämisestä ja luomisesta tehokkaampaa. JavaScript-kehykset kukoistavat yhden sivun sovelluksien eli SPA (Single Page App) toteuttamisessa, eli verkkosivustoissa, joissa koko sivusto haetaan yhdellä HTTP-pyynnöllä. Navigoitaessa uusia sivuja ei haeta palvelimelta vaan ne luodaan dynaamisesti edellä mainittua DOMia käyttäen. Yhden sivun sovelluksilla on etuna saumaton käyttäjäkokemus, kun ei tarvitse odotella palvelimelta sivuja. Sovelluksen pystyy silti jakamaan näennäisesti sivuihin HTML5 historyAPI:n avulla. Vuessa historyAPIn manipuloinnin toteuttaa Vue Router, jolle voidaan ilmoittaa, minkä URLin alta löytyy kukin sivuksi jaettava kokonaisuus. Angularin ja Emberin kautta tutuksi tulleet *templatet* ja ”viiksisyntaksi” sekä kokemus komponenttien rakentamisesta Reactilla yhdistyivät Vuessa, mikä teki siitä helposti käyttöönotettavan teknologian. Tässä luvussa käyn läpi Vuen perusteita.

3.1 Työkalut

Vue-projektin voi aloittaa kätevästi Vue CLI -komentorivityökalulla, jolla noviisikin pystyy alustamaan projekteja ilman syvällistä ymmärrystä siitä, mitä kaikkea tarvitaan optimaalisen sovelluksen luomiseen. Komentorivityökalulle annetaan create-komento, joka luo projektipohjan. Projektiin haetaan automaattisesti lukuisia valittuja kirjastoja node_modules-hakemistoon. Näihin kuuluvat muun muassa kirjastot, joita tarvitaan kääntämään vue-päätteiset tiedostot ja pyörittämään paikallisesti kehitysympäristöä. Edellisten lisäksi projektia luodessa voidaan valita lisäkirjastoja oman tarpeen mukaan, kuten kuvasta 4 näkyy.



Kuva 4. Vue CLI create -komennon valintaikkuna

Vuex on Vuen virallinen tilanhallintasysteemi eli keskitetyn tiedon varastoinnin mahdollistava kirjasto, minkä avulla dataa voidaan jakaa koko sovelluksen sisällä. Reactin kanssa vastaava on usein React Redux. Vue CLIilla luotu projektin pohja tarjoaa tarvittavat skriptit kehitysympäristön ajamiseen ja sovelluksen kääntämistä varten. Lisäskriptejä on helppo tehdä automaattisesti luotujen pohjalta, kuten esimerkiksi kääntöskripti testiympäristöä varten. Varsinaista konfigurointia tarvitsee muuten hyvin vähän, yleensä vain ympäristöön liittyvien muuttujien lisäämistä. Muuttujat lisätään env-tiedostoihin, esimerkiksi .env.developmentiin, jolloin muuttuja olisi käytössä vain kehittämisen aikana. Tavallinen käyttötapaus on rajapintojen URLien määrittäminen eri ympäristöille.

Selaimiin on saatavilla Vue-kehitystyökalu Vue.js devtools, joka helpottaa Vuella kehittämistä. Chrome-selaimessa työkalu löytyy konsolista vue-tabin takaa. Työkalu paljastaa sovelluksen komponentit puun muodossa, josta pystyy esimerkiksi tarkastelemaan komponenttien tiloja. Myös Vuex on integroitu työkaluun siten, että sieltä pystyy tarkastelemaan keskitetyn tilan, storen, muutoksia ja siihen johtaneita toimintoja.

3.2 Komponentit

Vuessa komponentin konsepti on erityisen tärkeä, sillä se mahdollistaa sovellusten jakamisen pieniin ja itsenäisiin osiin, joita voidaan käyttää uudelleen [4]. Komponenttien avulla laajatkin kokonaisuudet ovat helpommin hallittavissa, sillä komponentteja on helppo lisätä, muokata ja poistaa. Vuessa komponenttipohjaisuus tulee parhaiten esille yhden komponentin tiedostoista, joissa yhtä komponenttia vastaa yksi tiedosto. Komponentin rakenne, toiminta ja ulkonäkö on jaettu tiedostossa kolmeen eri tagiin. Template-tagin sisälle tulevat HTLM-merkinnät ja aiemmin mainittu viiksisyntaksi eli tupla-aaltosulut, joiden avulla esitetään komponentin dataa. Templaten sisään merkitään myös muita komponentteja kustomoiduilla HTML-tageilla, mutta ne täytyy rekisteröidä ensin components-määrittelyn sisällä, joka löytyy script-osiosta. Script-tagin sisällä on komponentin data ja toiminnallisuus, kuten esimerkiksi elinkaarifunktioita. Style-tagin alle tulevat tavallisesti CSS-määrittelyt, jotka voidaan myös scope-merkinnän avulla jättää komponentin sisäisiksi. Scope-merkinnän lisäksi voi myös määritellä, onko syntaksi scss-tyyppistä. Nämä tiedostot ovat vue-päätteisiä ja build-skripti pitää huolen niiden kääntämisestä selainystävällisempään muotoon.

Vue-sovellukset alkavat juuresta, joka on Vue-instanssi ja myös komponentti. Kaikki komponentit ovat Vue-instansseja, mutta juuri-instanssi ottaa vastaan hieman eri argumentteja [5]. Vue CLIilla alustetussa projektissa löytyy juuri-instanssi ja muutama esimerkkikomponentti. Esimerkkikoodissa 1 el ilmaisee, mihin elementtiin instanssi kiinnittyy, jolloin viiksisyntaksilla voidaan näyttää sille kuuluva viesti.

```
<html>
 <div id="app">
   {{ message }}
  </div>
</html>

<script>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
</script>
```

Esimerkkikoodi 1. Vuen juuri-instanssi

Komponentin logiikka on käytännössä JavaScript-objektissa, jossa ominaisuudet määritellään avain-arvo-pareina. Esimerkiksi komponentin data, eli muuttujat ja niiden arvot, määritellään datafunktion sisällä, jonka tulee aina palauttaa objekti. Komponentin sisällä (esimerkiksi metodeissa) viitataan omaan instanssiin kutsumalla aina *this*, minkä kautta päästään myös käsiksi edellä mainittuihin muuttujiin. Komponentti voi myös vastaanottaa muuttujia vanhemmalta propsien muodossa. Tämä data on yhdensuuntaista, jolloin kyseisiä propseja ei voi muokata. Tällöin lapsikomponentti ei voi vahingossakaan muokata vanhempansa tilaa. [6.]

Kehittäjien tarpeeseen päästä vaikuttamaan komponentin elinkaaren eri vaiheissa on vastattu elinkaarifunktioilla. Niitä käyttämällä kehittäjä voi tehdä toimenpiteitä tiettyihin aikoihin komponentin elinkaaressa, esimerkiksi ennen komponentin luomista ja komponentin luomisen jälkeen. Funktiot ovat seuraavat, kun aloitetaan elinkaaren alusta: beforeCreate, created, beforeMount, mounted, beforeUpdate, updated, beforeDestroy, destroyed. Kuten nimet antavat ymmärtää, tapahtumille löytyy aina pari: ennen ja jälkeen. Esimerkiksi created-funktiossa on tavallista tehdä HTTP-pyyntö, jolla noudetaan dataa sivulla näytettäväksi. Mounted-funktion kohdalla templaten merkinnät on tulkittu ja saatettu HTML-elementeiksi, joten ennen tätä vaihetta ei voida hakea elementtejä suoraan käsiteltäviksi. [5.]

Muista komponentin ominaisuuksista tärkeimpiä ovat methods, computed, watch, components, directives ja filters. Monet näistä voi rekisteröidä komponentin sisällä tai globaalisti. Komponentin metodit määritellään methods-objektiin avain-funktio-pareiksi. Funktioita ei tule määritellä nuolisyntaksilla, koska silloin ne eivät kiinnity komponentin

instanssiin [5]. Metodit yleensä sisältävät perustoiminnallisuudet kuten painikkeisiin liitettävät onclick-funktiot, kuten esimerkkikoodissa 2.

```
methods: {
  handleClick: function() {
     console.log("Clicked button")
  }
}
```

Esimerkkikoodi 2. Esimerkkimetodi

Computed-arvot tarjoavat mahdollisuuden yhdistellä eri muuttujia reaktiivisiksi arvoiksi. Tämä vähentää tarvetta tehdä operaatioita suoraan templatessa, mikä paisuttaisi koodia ja tekisi siitä monimutkaista etenkin silloin, kun haluaisi käyttää samaa arvoa useamman kerran. Computediin määritellään siis funktioita, jotka palauttavat aina arvon. Arvo on yleensä johdettu yhdestä tai useammasta komponentin sisältä löytyvästä muuttujasta, joiden muuttuessa tämä funktio reagoi siihen ja palauttaa uuden arvon. Esimerkiksi käyttäjän syöttämien numeroiden summan voisi määritellä computed-arvoksi, kuten esimerkkikoodissa 3. [7.]

```
computed: {
  sum: function() {
     return this.firstInput + this.secondInput
  }
}
```

Esimerkkikoodi 3. Computed-esimerkki

Vuen watcherit luovat computedin lisäksi mahdollisuuden reagoida muuttujien muuttuessa. Usein olisi houkuttelevaa käyttää watcheria computedin sijaan, mutta watcherit kannattaa säästää asynkronisia ja raskaita operaatioita varten. Watchin alle annetaan avaimiksi komponentista löytyvien arvojen nimiä, esimerkiksi edellä mainittu computed-arvo. Avaimen arvoksi tulee funktio, jolla on kaksi parametria, ensimmäinen uudelle arvolle ja toinen vanhalle. Funktio ajetaan aina kun katseltavassa arvossa on tapahtunut muutos. Käyttötarkoitus watcherille on esimerkiksi syötteen validoiminen, kuten esimerkkikoodissa 4, jossa tutkitaan uuden arvon tyyppi. [8.]

```
watch: {
  firstInput: function(oldVal, newVal) {
   this.firstInputIsValid = typeof newVal === "number"
   }
}
```

Esimerkkikoodi 4. Syötteen katselija

Komponenttien on tarkoitus olla koostettavissa myös muista komponenteista, jotka on haettu JavaScriptin moduulisysteemin avulla, eli käyttämällä importtia. Haetut komponentit tulevat objekteina, jotka voidaan suoraan lisätä isäntäkomponentin components-määrityksen alle. Komponentin voi myös luoda suoraan Vuen components-funktion avulla. Tämä rekisteröi komponentin käytettäväksi isäntäkomponentin templates-osioon, jossa sen voi lisätä muun HTML-merkintöjen ohessa.

```
<script>
import ExampleComponent from "@/components/ExampleComponent.vue"

export default {
  name: "ParentComponent",
  components: {
     ExampleComponent
  }
}
</script>
```

Esimerkkikoodi 5. Komponentin rekisteröiminen

Tavan muokata tekstiä suoraan template-osiossa, ilman computed-arvojen luomista jokaiselle muuttujalle, tarjoavat filtterit. Filtteri on funktio, joka ottaa arvon vastaan ja palauttaa sen muokattuna. Filtterit rekisteröidään komponentin filters-määrityksiin lisäämällä tuodun filtterin tai luomalla suoraan riville. Filtereitä voi käyttää templatessa muuttujien kanssa Unixista tutulla putkitussyntaksilla, kuten esimerkkikoodissa 6 näkyy. [9.]

```
<template>
 <p>{{ someString | capitalize }}</p>
</template>

filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
```

Esimerkkikoodi 6. Esimerkkifiltteri

3.3 Direktiivit

Direktiivit ovat HTML-elementtiin tai komponenttiin kytkettäviä merkintöjä, jotka määrittävät käyttäytymistä. Direktiivejä voi rekisteröidä tai määritellä itse komponentissa direktiivien alle. Esimerkiksi VueDraggable kirjasto tuo käytettäväksi v-drag-and-drop-direktiivin, joka tekee elementistä liikuteltavan. Vuelle valmiita direktiivejä löytyy lukuisia, joita käyn läpi tässä luvussa.

Kaksisuuntainen kytkentä tarkoittaa synkronisointia näkymän ja datan välillä. Vuen kaksisuuntainen kytkentä onnistuu käyttämällä v-model-direktiiviä. Direktiivin voi lisätä mihin tahansa syöte- tai valintakenttään, jolloin komponentin data kytketään siihen kaksisuuntaisesti. Tällöin datan muutokset komponentin sisällä heijastuvat kytketyssä kentässä, ja kenttään syötetty data näkyy komponentin sisällä.

```
<input type="text" v-model="foo">
```

Esimerkkikoodi 7. Kaksisuuntainen kytkentä v-model-direktiivillä

Listojen luomista dynaamisesti helpottaa templatessa käytettävä silmukkadirektiivi v-for. v-for luo siihen kytkettyä elementtiä jokaisen silmukan iteraation yhteydessä. Direktiiville annetaan taulukko ja nimetään yksikkö ja avain, jolloin niitä voi käyttää elementissä. Alkion käyttö on rajoitettu silmukkaelementin sisälle. Käyttökohteita ovat listat ja taulukoiden rivit. template-tagiin yhdistettynä voidaan jokaisella iteraatiolla luoda esimerkiksi kaksi tr-elementtiä, eikä template-tagista jää HTML-merkintää. Tämä on hyödyllinen, kun halutaan luoda table-elementille kaksi riviä per iteraatio, mutta ei haluta kääriä niitä mihinkään elementtiin. Taulukon sisältäessä objekteja, käsitellään niiden avaimia tavalliseen tapaan. [10.]

```
<template v-for="item in items">
     <div>{{ item.foo }}</div>
     <div>{{ item.bar }}</div>
</template>
```

Esimerkkikoodi 8. v-for-silmukka objekteilla

Oleellista dynaamisen näkymän luomisessa on myös templatessa käytettävä ehtolause-direktiivi v-if. v-if toimii konditionaalin tavoin ja luo siihen kytketyn elementin, jos direktiiville annettu ehto täyttyy. Tämän jälkeen tuleville elementeille voi antaa v-else- ja v-if-else-direktiivejä, jotka käyttäytyvät kuin else-if-ketju yleensä. Useamman elementin voi kääriä template-elementin sisälle, jos haluaa ehdollistaa koko ryhmän kerralla. Direktiiville voi antaa suoraan arvon tai laskea arvon rivillä. Myös funktio, joka palauttaa arvon, toimii. v-show toimii kuin v-if, mutta elementti piilotetaan CSS display -määrityksen avulla. v-else ei toimi v-shown kanssa. [11.]

```
<p v-if="luku < 10">
      Luku on alle 10.
</p>
<p v-else-if="luku > 20">
     Luku on yli 20.
</p>
<p v-else>
     Luku on 10 ja 20 välissä.
</p>
```

Esimerkkikoodi 9. v-if-else-ketju

Tapahtumankuuntelijoita tarvitaan kuuntelemaan käyttäjän tekemiä toimintoja verkkosivulla, jotta niihin voitaisiin reagoida. v-on-direktiivi tarjoaa tavan lisätä tapahtumankuuntelijoita helposti. v-on-direktiiville annetaan ensin tapahtuma, jota kuunnellaan, ja parametriksi, mitä tehdään kyseisen tapahtuman sattuessa. Tapahtumia ovat esimerkiksi painallukset (hiiri tai näppäin), vierittäminen ja tai muut itse luodut tapahtumat, joita voi lähettää komponentista käsin. Rivillä direktiiville määritetään tapahtuman tyyppi esimerkiksi hiiren painallus, joka lisäisi laskurin lukua. v-on-merkinnän voi vaihtoehtoisesti korvata @-merkillä. Näppäinten painallukset otetaan vastaan keyup-määrittelyllä, jolle voi antaa myös näppäimen tai näppäinyhdistelmän, johon reagoidaan. Tapahtuman voi ottaa myös talteen $event-muuttujan avulla esimerkiksi funktioon, joka voi esimerkiksi tutkia, mikä näppäin päästettiin ylös. Esimerkkikoodissa 10 näkyy edellä mainittuja tapahtumankuuntelijoita. [12.]

```
v-on:click="laskuri++"
@keyup.alt.enter="laskuri++"
@keyup="painalluksenTutkija($event)"
```

Esimerkkikoodi 10. v-on-syntaksia

Lapsikomponentin kommunikoinnin vanhempansa kanssa mahdollistaa emit-funktio. Emit-funktiolla lähetetään nimetty tapahtuma, jonka komponentin vanhempi ottaa vastaan. Funktion ensimmäiseksi parametriksi tulee tapahtuman nimi ja toiseksi, mitä halutaan lähettää, esimerkiksi arvo. Jos haluaa lähettää useamman arvon kerrallaan, tulee ne laittaa taulukon tai objektin sisään. Vanhempi ottaa tapahtuman vastaan kuuntelemalla tapahtumalle määriteltyä nimeä, joka annetaan direktiiville. Esimerkkikoodissa 11 on emit-funktion kutsuminen sekä vanhemman näkökulma lapsikomponentista, johon on lisätty kuuntelija. [13.]

```
this.$emit('tapahtumanNimi', arvo)
<lahettaja-komponentti v-on:tapahtumanNimi="uusiArvo = $event">
```

Esimerkkikoodi 11. Tapahtuman lähettäminen ja vastaanottaminen

v-bind helpottaa tekemään HTML-elementtien ominaisuuksista dynaamisia, kuten esimerkiksi class- ja style-määrittelyistä. V-bindille annetaan ominaisuus ja mikä arvo siihen sidotaan, kuten esimerkkikoodissa 12 näkyy.

Style-ominaisuudelle tulee antaa tyyliobjekti, joka käytännössä näyttää samalta kuin CSS-luokka. Tyyliobjektin avaimien tulee vastata CSS-määreitä, jos CSS-määreessä on väliviiva, tulee se korvata siten, että väliviiva poistetaan ja sen jälkeen tuleva sana alkaa isolla kirjaimella (camelcase). Määreen voi asettaa myös heittomerkkien sisään, jos siitä löytyy väliviiva. Tämä ja edellinen merkkaustapa näkyvät esimerkkikoodissa 12. Arvot objektissa tulee olla stringejä.

```
<div v-bind:class="cssLuokka">

<div v-bind:class="{'luokka-1':arvo===1, 'luokka-2':arvo===2}">

tyyliObjekti: {
     'border-top': 'solid black 1px',
     fontSize: '16px'
}
```

Esimerkkikoodi 12. v-bind-syntaksia ja tyyliobjekti

Muita hyödyllisiä käyttökohteita on dynaaminen title ja placeholder, jotka ovat erityisen tärkeitä, kun käytössä on useampia kieliä. Esimerkkikoodissa 13 on Vuex i18n -kirjaston funktio, joka palauttaa käytössä olevan kielen tekstin. V-bind-etuliitteen voi jättää pois.

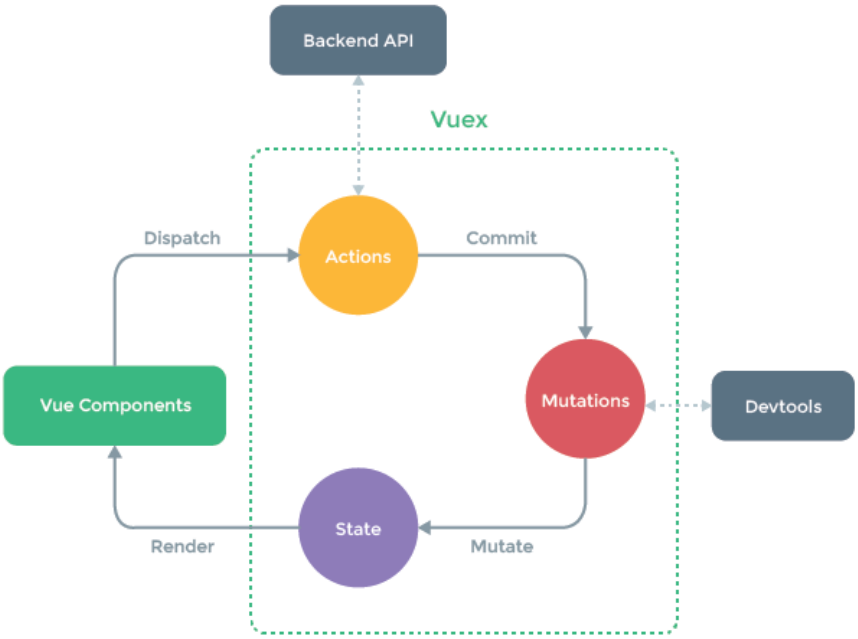
```
<input :placeholder="$t('kenttaTeksti')" :title="$t('ohjeTeksti')">
```

Esimerkkikoodi 13. Dynaaminen placeholder ja title

3.4 Vuex

Sovelluksilla on yleensä tarve saada tallennettua sellaista tietoa, joka on kaikkien komponenttien käytössä, kuten esimerkiksi käyttäjän tiedot. Pienessä sovelluksessa on vielä mahdollista tallettaa sovelluksen yhteiset tiedot juuritasolle, josta ne voi valuttaa lapsikomponentista toiselle. Sama pätee myös yhteisen tilan muuttamiseen, jolloin käsky tilan muuttamiseen tulee siirtää komponentilta toiselle, aina juureen saakka. Sovelluksen kasvaessa isommaksi ja syvemmäksi yhteinen tila paisuu ja tilanhallinnasta tulee haastavaa. Silloin tulee tarve tilanhallintasysteemille.

Vuen virallinen tilanhallintasysteemin kirjasto on Vuex. Se toimii pitkälti samalla tavalla kuin muutkin vastaavat kirjastot siten, että sillä on tila, josta voi hakea tietoa ja päivittää sitä lähettämällä (dispatch) toimintoja (actions). Vuexilla luodaan store, joka rekisteröidään sovelluksen juuri-instanssiin, mikä on yksi tapa, jolla juuri poikkeaa muista komponenteista. Kun store on rekisteröity, siihen pääsee käsiksi globaalin store-muuttujan kautta kaikista komponenteista. Storen saa käyttöön myös JavaScript-moduuleissa hakemalle sen. Actionit ja siihen yhdistyvä tilanhallinta luovat yhdessä kerroksen sovellukselle, jonka kautta on luonnollista kommunikoida rajapinnan kanssa. Tämän syitä käyn läpi tuonnempana. Kuvassa 5 näkyy Vuexin suhde komponentteihin, rajapintaan ja devtools-työkaluun, joka on selaimeen hankittava lisäosa. [14.]



Kuva 5. Vuex-kaavio [14]

Storen tila eli state määritellään etukäteen ja on tavallinen JavaScript-objekti, aivan kuten tavallistenkin komponenttien tilat. Storen tilaan pääsee käsiksi suoraan globaalin muuttujan kautta, mutta kun tila kasvaa syvemmäksi, tulee tarve gettereille. Sen sijaan, että käytäisiin läpi pitkä rimpsu sisäkkäisiä objekteja, voidaan laittaa getteri tekemään se. Getterissä voidaan myös muokata palautettavaa tietoa, esimerkiksi yhdistää kaksi eri tietoa ja palauttaa ne yhtenä. Getterin voi yhdistää komponentin sisällä computed-arvoon, jolloin voidaan määrittää myös setteri, joka lähettää uuden tiedon storelle actionin avulla, joka käydään läpi tuonnempana. Getterit määritellään storessa getters-objektiin. Koodiesimerkissä 14 on storen määrittely, jolle kuuluu myös getteri, joka palauttaa tehdyt askareet, sekä getterin käyttö komponentissa. [15.]

```
/* Store */
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})

/* Komponentti */

<div>Count: {{ doneTodosCount }} </div>
///
computed: {
  doneTodosCount () {
    return this.$store.getters.doneTodosCount
  }
}
```

Esimerkkikoodi 14. Storen määrittely getterin kanssa ja getterin käyttö komponentissa [15]

Mutaatiot mahdollistavat tilan muokkaamisen synkronisesti. Vue devtools ottaa tilasta aina kuvan ennen ja jälkeen mutaatiota, jolloin tilaa pystyy seuraamaan, mikä taas helpottaa bugien etsimistä. Tämän takia mutaatiossa ei saa tapahtua asynkronisia operaatioita, kuten esimerkiksi HTTP-pyyntö. Mutaatiot määritellään storen sisälle mutations objektiin ja ovat tyypiltään funktioita, jotka saavat ensimmäiseksi parametriksi tilan ja toiseen parametriin voi sijoittaa lisättävän tiedon. Avaimeksi annetaan mutaation nimi. Mutaatiota kutsutaan käyttämällä storen commit-funktiota, jolle annetaan ensimmäiseksi parametriksi mutaation nimi ja toiseksi arvo. Koska commit ottaa vastaan vain kaksi parametria, pitää toisesta parametrista tehdä objekti, jos haluaa antaa mutaatiolle useampia arvoja kerralla. Reaktiivisuuden kannalta tilaan tai siinä sijaitseville objekteille ei kannata määrittää uusia avaimia suoraan, vaan Vuen set-funktiolla, tai korvaamalla vanha objekti uudella. [16.]

```
mutations: {
  increment (state, n) {
    state.count += n
  }
}
```

Esimerkkikoodi 15. Mutaation määrittely storessa

```
this.$store.commit('increment', 5)
```

Esimerkkikoodi 16. Commit-kutsu

Actionit ratkaisevat asynkronisuusongelman mutaatioiden kanssa. Actioneiden sisällä voidaan tehdä asynkronisia HTTP-pyyntöjä palvelimelle ja kutsua commit-funktiota pyynnön ollessa valmis. Tällöin rajapintakutsut sijoittuvat luonnollisesti actioneiden sisälle. Actioneiden ei ole kuitenkaan pakko käyttää commit-funktiota, vaan ne voivat myös palauttaa datan suoraan komponentille, mutta koska ne ovat storessa, niitä voi kutsua kaikkialta. Actionit tulevat storeen actions-objektiin ja ovat hyvin samanlaisia mutaatioiden kanssa. Erona on kuitenkin ensimmäinen parametri, joka on kontekstiobjekti staten sijaan. Actionin määrittely näkyy esimerkkikoodissa 17. Kontekstista otetaan kuitenkin yleensä vain commit-funktio talteen destrukturoinnin avulla. Toisena parametrina action saa vapaavalintaisen dataobjektin, aivan kuten mutaatiot. Actioneja kutsutaan komponenteista käsin vain storen dispatch-metodin avulla. Käyttö on pitkälti sama kuin commitissa, dispatchin ensimmäiseksi parametriksi tulee actionin nimi ja toiseksi dataobjekti, mikä näkyy esimerkkikoodissa 17. Actionin voi yhdistää komponentissa getterin kanssa computed-arvoon, jolloin voidaan luoda kaksisuuntainen kytkentä suoraan storen arvoon. Tämä näkyy esimerkkikoodissa 18. [17; 18.]

```
actions: {
  increment ({ commit }, n) {
    commit('increment', n)
  }
}

this.$store.dispatch('increment', n)
```

Esimerkkikoodi 17. Actionin määrittely ja käyttö komponentista käsin

```
<input v-model="message">
…
computed: {
  message: {
    get () {
      return this.$store.getters.message
    },
    set (value) {
      this.$store.commit('updateMessage', value)
    }
  }
}
```

Esimerkkikoodi 18. Syötekentän kaksisuuntainen kytkentä suoraan storeen

3.5 Router

Vuella tehty sovellus olisi täysin mahdollista rakentaa yhdelle sivulle siten, että näkymää muutetaan dynaamisesti käyttäjän toimintojen mukaan. Tällöin käyttäjä kuitenkin hävittäisi sisäisen tilan, kun hän päivittäisi sivun. Käyttäjä ei myöskään pystyisi menemään taaksepäin tilassa. Näkymien liittäminen eri URLeihin poistaisi tämän ongelman sekä tarjoaisi linkittämisen suoraan eri näkymiin. Liittämisen tarjoaa Vuelle Vue Router, joka on Vuen virallinen reititinkirjasto, joka mahdollistaa URLiin perustuvan navigoinnin. Reitin rekisteröidään sovelluksen juuri-instanssiin samalla tavalla kuin Vuex storekin. Reitittimen kanssa juurikomponenttiin tulee sijoittaa router-view-komponentti, joka luo sisälleen sen komponentin, mihin sen hetkinen URL on liitetty. Liitosmäärittelyt annetaan reitittimelle taulukkomuodossa, johon on listattu objekteina URL ja komponentti, mikä sille kuuluu, kuten esimerkkikoodissa 19. [19.]

```
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User }
  ]
})
```

Esimerkkikoodi 19. Routerin määrittely

Router-view komponentin lisäksi käyttöön tulevat router-link-komponentit, jotka toimivat linkkeinä. Router-linkille määritetään kohde to-määrittelyn avulla, jolle voidaan antaa suoraan URL tai polkuobjekti. Polkuobjektissa voi olla string-tyyppinen path tai name, joka viittaa polun nimeen. Esimerkki kummastakin linkittämisen tavasta näkyy koodiesimerkissä 20.

```
<router-link to="/user/5">
<router-link :to="{ name: 'user', params: { id:5 } }">
```

Esimerkkikoodi 20. Linkkejä

Routeria pystyy myös manipuloimaan komponentista käsin globaalin reititinmuuttujan avulla. Navigointi onnistuu silloin kutsumalla reitittimen push-funktiota, jolle annetaan samanlainen objekti kuin linkille. Tällöin voidaan esimerkiksi siirtää käyttäjä toiseen näkymään. [20.]

Polkumäärittelyissä voi määritellä myös navigation guard -funktioita, jotka toimivat välikerroksena navigointitapahtumille ja varmistavat, onko käyttäjällä oikeus siirtyä näkymään. Guard-funktio saa parametreikseen, minne ollaan menossa, mistä ollaan tulossa, ja callbackin, jolla voidaan jatkaa siirtymistä tai evätä se. [21.]

4 Golang

Palvelinpuoli eli back-end päädyttiin toteuttamaan vähemmän tunnetummalla Golangilla (Go), joka on Googlen kehittämä ohjelmointikieli. Sovellukset ovat nopeita kääntää ja siirtää kohdejärjestelmään ajettavaksi verrattuna Java-pohjaisiin ratkaisuihin, jotka vaativat runsaasti konfigurointia ja saattavat vaatia myös Apache Tomcatin pyöriäkseen.

4.1 Aloitus

Golla työskenteleminen aloitetaan lataamalla ja asentamalla avoimen lähdekoodin jakelu, jolla pystytään kääntämään, ajamaan ja testaamaan Go-sovelluksia. Go-projektit luodaan osoitettuun työtilaan. Oletustyötila on käyttäjän kotihakemistossa sijaitseva go-hakemisto. Go-hakemiston sisällä projektit luodaan src-kansion alle. Jos työtila ei ole kotihakemistossa, joutuu sen polun lisäämään GOPATH-muuttujaan. Tällöin työtilassa pystyy käyttämään Go-komentorivityökalua. [22.]

Projektit jaetaan pakkauksiin (package), jotka tulee myös määritellä tiedoston alussa. Jokainen pakkaus on omassa hakemistossaan. Projekteilla tulee olla main tiedosto, joka nivoo koko projektin yhteen ja kuuluu main-pakkaukseen, kuten esimerkkikoodissa 21 näkyy ensimmäisellä rivillä. Projekti rakennetaan antamalla komentorivillä go build -komennolle main-tiedosto. Tällöin ilmestyy main-tiedosto, jonka voi ajaa suoraan komentoriviltä. Kirjastot ja paketit haetaan käytettäväksi importilla. Fmt-kirjasto kuuluu peruskirjastoihin, joka tulee asennuksen mukana ja sillä voi muun muassa printata tekstiä, mikä näkyy myös esimerkkikoodissa 21. [23.]

```
package main

import "fmt"

func main() {
     fmt.Printf("hello, world\n")
}
```

Esimerkkikoodi 21. Hello world Golangilla

4.2 Muuttujat

Golang muistuttaa hieman C-ohjelmointikieltä, sillä se on staattisesti tyypitetty ja käytössä on osoittimet ja structit. Golangista löytyy kuitenkin roskien keruu eikä muistin käytön kanssa tarvitse olla yhtä varovainen, vaikka muistivuoto onkin mahdollista. Muuttujan tyyppiä ei tarvitse määrittää etukäteen, jos käytetään lyhyttä määrittelytapaa. Golangia pystyy kokeilemaan suoraan selaimesta käsin osoitteessa play.golang.org. [24]

Nil on golangin nolla-arvo, eli se vastaa null-arvoa muissa kielissä, muutamalla erotuksella. Nil-arvoa ei tarvitse määritellä etukäteen ja sen voi saada esimerkiksi tyhjät taulukot, mapit, osoittimet ja funktiot [25]. Perustyypit kuten string ja int eivät voi olla nil, mikä vaikeuttaa yhteensopivuutta JSONin kanssa esimerkiksi silloin, kun otetaan vastaan JSON-muotoista dataa, joka voi sisältää null-arvoja. Osoittimet voivat kuitenkin olla nil, joten silloin voidaan päästä lähemmäksi yhteensopivuutta. Tähti-merkki tyypin edessä merkitsee, että muuttuja on sen tyypin osoitin. &-merkillä voi viitata muuttujan muistipaikkaan ja laittaa osoittimen osoittamaan sitä, mikäli se on oikeaa tyyppiä. Esimerkkikoodissa 22 muuttuja k määrittyy int-tyyppiseksi lyhyellä määrittelytavalla, joka on siis kaksoispiste yhdistettynä yhtäsuuruusmerkkiin. Muuttuja i määritellään tavallisella tavalla, jonka jälkeen osoitin p laitetaan osoittamaan siihen. Muuttamalla muuttujaa i myös osoitin osoittaa uutta arvoa.

```
func main() {
      k := 3 //lyhyt määrittely
      fmt.Println(k)


      var i int = 1
      var p *int

      p = &i
      fmt.Println(*p)
      i = 4
      fmt.Println(*p)
}
```

Esimerkkikoodi 22. Pitempi ja lyhyempi int-muuttujan määrittely sekä osoitin

4.3 Struct-tyyppi

Struct on kokoelma kenttiä, joka tulee määritellä funktioiden ulkopuolella. Structille määritellään nimi ja sen kentille nimet ja tyypit. Tyypit voivat olla myös osoittimia. Kenttien nimet tulee alkaa isolla kirjaimella tai kenttään ei pääse viittaamaan. Struct on tavallisesti JSONin vastinkappale, eli asiakaspäästä tulevat JSON-muotoiset pyynnöt käännetään structeiksi ja samaten myös toiseen suuntaan. Myös tietokannasta haetut relaatiot on helppo kääntää structeiksi. Structit eivät kuitenkaan ole yhtä joustavia kuin JSONit eikä niihin voi määritellä uusia kenttiä lennosta. Structeille pystyy myös määrittämään tageja, jotka auttavat relaation tai JSONin kääntämisessä structiksi. Ilman tageja tietokannan sarakkeen tai JSONin kentän tulisi olla saman niminen kuin structin vastaava, mikä aiheuttaa ongelmia, sillä usein data on nimetty eri tavalla tietokannassa tai JSONissa. Tagit määritellään structiin antamalla lähde ja kentän nimi. Anonyymit structit voidaan määritellä suoraan koodissa esimerkiksi kertakäyttöisiksi. [26.]

Esimerkkikoodissa 23 on määritelty Dog-struct, jolla on nimi, paino ja onko hyvä poika -kenttä. Kentille on määritelty vastaavat tietokanta (db) ja JSON -kenttien nimet tageilla, mikä osoittaa eri nimeämiskäytännöt tietokannan ja JavaScriptin maailmassa.

```
type Dog struct {
     Name      string `db:"name" json:"name"`
     Weight    int    `db:"weight" json:"weight"`
     IsGoodBoy bool   `db:"is_good_boy" json:"isGoodBoy"`
}
```

Esimerkkikoodi 23. Structin määrittely tagien kanssa

Esimerkkikoodissa 24 yllä oleva Dog-struct on otettu käyttöön ja siitä on luotu Musti- ja Pekka-instanssi. Musti on luotu suoraan lyhyellä määrittelyllä, kun taas Pekan kentät on täytetty yksi kerrallaan. Myös Mustin kenttiä muutettaisiin jälkeenpäin samalla tavalla.

```
func main() {
	musti := Dog{"Musti", 5, true}

	var pekka Dog
	pekka.Name = "Pekka"
	pekka.Weight = 7
	pekka.IsGoodBoy = false
}
```

Esimerkkikoodi 24. Structin käyttö

Esimerkkikoodissa 25 Dog-struct ensin määritellään anonyymisti ja luodaan siitä instanssi saman tien.

```
func main() {
	anonymousStruct := struct {
		Name string
		Weight int
	}{
		"Musti",
		5
	}
	fmt.Println(anonymousStruct.Name)
}
```

Esimerkkikoodi 25. Anonyymi struct

4.4 Map-tyyppi

Map on kokoelma avain-arvo-pareja. Mapeille määritellään avaimen tyyppi ja arvon tyyppi [27]. interface-tyyppistä arvoa käyttäen mapista saa yhtä joustavan kuin JSON-objektista, koska interface-arvo voi olla mitä tyyppiä tahansa. Structit ovat kuitenkin se, mitä käytetään tiedon esittämiseksi tietokannasta. Esimerkkikoodissa 26 luodaan aikasempi Dog-struct map-tyyppinä. Make-funktiolle annetaan taulukon tyyppi, pituus ja viimeiseksi parametriksi voisi antaa taulukon kapasiteetin, jota ei tässä määritellä. Funktio luo tyhjän mapin, jonka jälkeen sille määritellään kenttiä käyttäen stringejä avaimina, ja koska mahdolliset arvot oli määritelty interface-tyypiksi, voidaan arvoiksi antaa mitä vaan.

```
func main() {
	m := make(map[string]interface{}, 0)
	m["Name"] = "Musti"
	m["Weight"] = 5
	m["IsGoodBoy"] = true
	fmt.Println(m["Name"])
}
```

Esimerkkikoodi 26. Make-funktioilla luotu tyhjä map, jolle avaimen tyypiksi määritelty string ja arvoksi tyhjä interface

4.5 Funktiot

Funktiot toimivat Go:ssa pitkälti samalla tavalla kuin muillakin kielillä, mutta vaativat aina palautuksen, paitsi main-funktio. Funktiot voivat palauttaa useamman arvon kerralla. Usein yksi arvo on varattu virheille. Kaikki funktiolle määritellyt palautustyypit tulee kuitenkin palauttaa joka kerta, jopa virhetilanteissa. Luonnollista olisi tällöin palauttaa tyhjä arvo virheen lisäksi, mutta koska esimerkiksi struct ei voi olla nil, tulee se palauttaa vaikka sitten alustamattomana. Tämän voi kuitenkin kiertää laittamalla palautustyypiksi pointterin, jolloin voidaan palauttaa tyhjä pointteri. Funktioita voi myös kiinnittää structeihin, mikä tekee niistä structien metodeja [28]. Esimerkkikoodissa 27 näkyy summausfunktio. Funktion parametreille määritellään tyypit ja funktion lopussa määritellään palautettava tyyppi.

```
func sum(a int, b int) int {
	return a + b
}
```

Esimerkkikoodi 27. Summa-funktio.

Esimerkkikoodissa 28 määritellään metodi aikaisemmin määritellylle Dog-structille. Metodi määritellään antamalla sille ensimmäiseksi struct, jolle metodi kuuluu. Metodille on määritelty kaksi palautusarvoa: ensimmäinen on string-osoitin ja jälkimmäinen error-tyyppiä. Metodissa Dog haukkuu, eli palauttaa string-osoittimen ja tyhjän virheen, jos se on hyvä poika. Muussa tapauksessa se palauttaa tyhjän string-osoittimen ja virheen. Haukunta palautetaan string-pointterina, koska silloin, jos ei haukuta, voidaan palauttaa nil-arvo, kuten näkyy jälkimmäisessä palautuslauseessa. Virhe on määritelty suoraan palautukseen errors-kirjaston avulla.

```
func (d Dog) Bark() (*string, error) {

      if (d.IsGoodBoy) {
         bark := "Woof!"
         return &bark, nil
      }
      return nil, errors.New("Error no bark found")

}
```

Esimerkkikoodi 28. Dog-structin metodi, joka palauttaa string pointerin ja errorin.

Esimerkkikoodissa 29 määritellään Musti ja kutsutaan haukuntametodia. Palautukset talletetaan muuttujiin bark ja err. Virhe tarkistetaan tarkistamalla, onko err tyhjä eli nil, jos err on nil, tiedetään, ettei virhettä tapahtunut.

```
func main() {
      musti := Dog{"Musti", 5, true}

      bark, err := musti.Bark()
      if err != nil {
            fmt.Println(err)
            return
      }
      fmt.Println(*bark)
}
```

Esimerkkikoodi 29. Funktion palautus ja virheen tarkistus.

4.6 Kirjastot

Kirjastoja haetaan go get -komennolla. Kirjastot tulevat samaan hakemistoon kuin projektitkin eli go/src. Komennolle annetaan url, jossa kirjasto sijaitsee, usein Github-versionhallintapalvelussa. [29.]

Toimittajarahoituksessa Go-sovelluksen toimiessa palvelimena tärkeimmät kirjastot liittyvät tietokantoihin ja palvelimena toimimiseen. Gorillan mux (github.com/gorilla/mux) toimii reitittimenä, jonka avulla voi määrittää palvelimen päätepisteet, joihin HTTP-pyyntöjä voi kohdistaa. Muxin kanssa toimii Gorillan handlers-kirjasto (github.com/gorilla/handlers), jonka avulla voi luoda käsittelijöitä kyseisille päätepisteille. Tietokantayhteyden sai Go:n tietokanta-ajureilla ja kutsujen tekemistä helpotti Jmoironin sqlx (github.com/jmoiron/sqlx). Runsasta SQL-lauseiden kirjoittelua helpotettiin jälkeenpäin Volatiletechin SQLBoiler-kirjastolla (github.com/volatiletech/sqlboiler), joka toimi ORMina.

SQLBoiler ei kuitenkaan ole yhtä rikas toiminnoiltansa kuin Javan Hibernate tai muiden kielten vastaavat toteutukset. SQLBoiler vaatii myös työkalun käyttöä, jolla koodiin generoidaan mallit tietokantataulujen pohjalta.

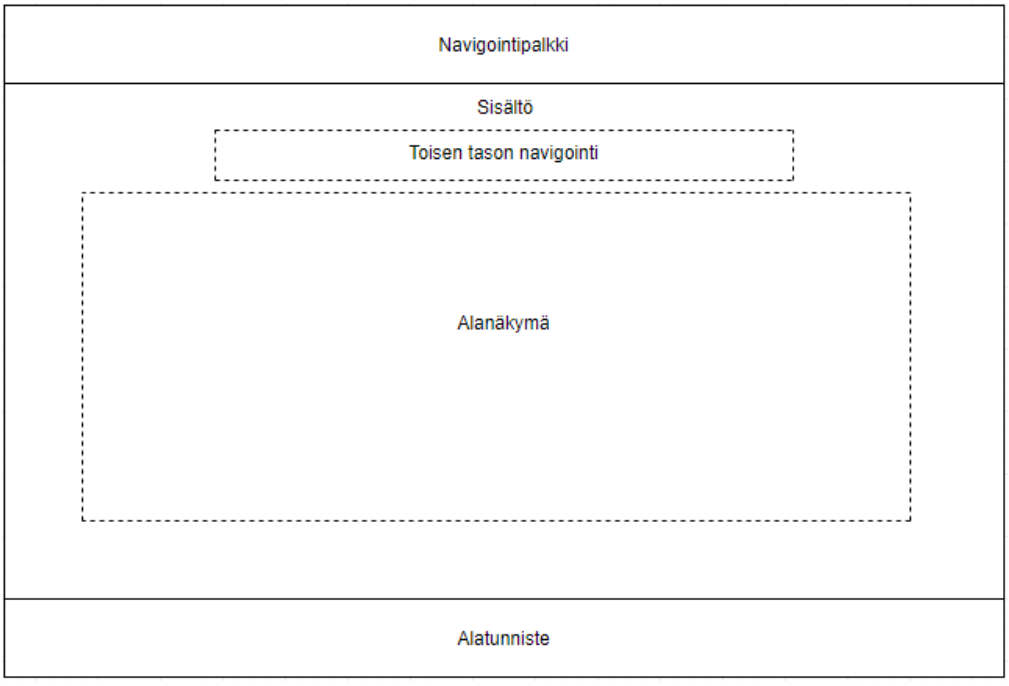
Virheiden etsimistä helpottaa ja pyyntöjä tallettaa Sirupsenin Logrus-kirjasto (github.com/sirupsen/logrus), joka tekee lokimerkintöjen luomisesta helppoa.

5 Toteutus

Tässä luvussa käydään läpi valmistuneita näkymiä ja muutamia niissä ilmenneitä ongelmia ja ratkaisuja sekä hieman rajapintatoteutusta. Tähän kuuluvat vain tärkeimmät näkymät, sillä perusnäkymät kuten käyttäjäasetukset tai kirjautumisikkuna ovat tavanomaisia eivätkä sisällä mielekkäitä ongelmia. Tekijän panos rajapinnassa oli vähäisempi kuin käyttöliittymässä.

5.1 Päätaso

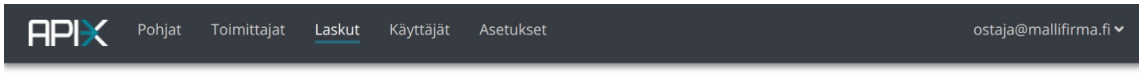
Päätasolla toimittajarahoitus on jaettu ylänavigointiin, sisältöön ja alatunnisteeseen. Niissä näkymissä, missä on alanäkymä, sisältö jakautuu alemman tason navigointiin sekä alanäkymään. Näkymän rakennetta on hahmoteltu kuvassa 6.



Kuva 6. Käyttöliittymän rakenne

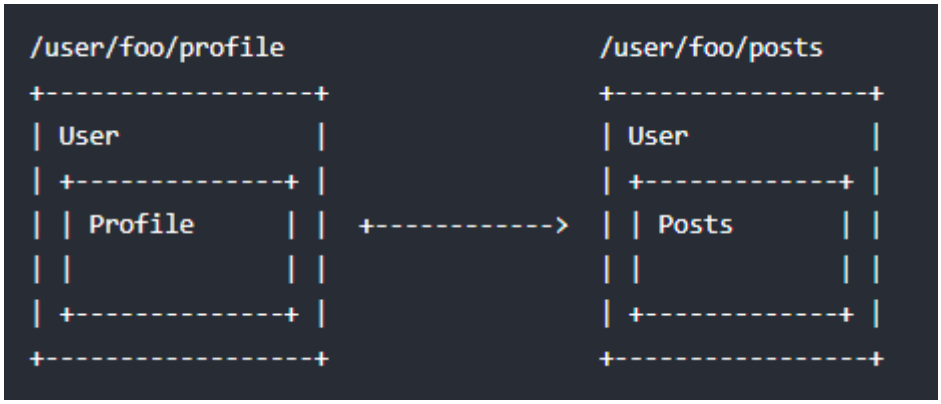
Toimittajarahoituksen navigointiin valittiin yksinkertainen ylänavigointi. Ylänavigoinnissa jokaista kohdetta vastaa yksi otsake tai painike. Navigointityyliä valitessa otettiin huomi-

oon sovelluksen pääasiallinen käyttö työpöytäympäristössä. Joskin ylänavigointi soveltuu myös hyvin mobiiliympäristöön, kun se siirretään painikkeen taakse alas putoavaksi valikoksi. Ylänavigointi vie vain vähän tilaa sivuston ylälaidasta, ja se on aina näkyvissä. Ongelmia voi ilmetä, jos osioita on niin paljon, etteivät ne mahdu koko ruudun leveydelle edes tietokoneen näytöllä. Tämä ei kuitenkaan ole ongelma toimittajarahoituksessa, sillä osioita on vielä vain vähän. Ylänavigoinnin linkkeinä toimivat aikaisemmin kuvakkeet, jotka tullessaan kosketuksiin hiiren kursorin kanssa paljastivat osion nimen. Tämän kuitenkin todettiin hankaloittavan käyttäjää, kuten myös kuvakkeista valuvat valikot. Navigoinniksi jäivät siis vain osioiden nimet, jotka alleviivauksin ja fonttimuutoksin ilmaisevat käyttäjän senhetkisen sijainnin. Alanavigoinniksi jäivät saman tyyliset otsakkeet siellä, missä niitä käytettiin.

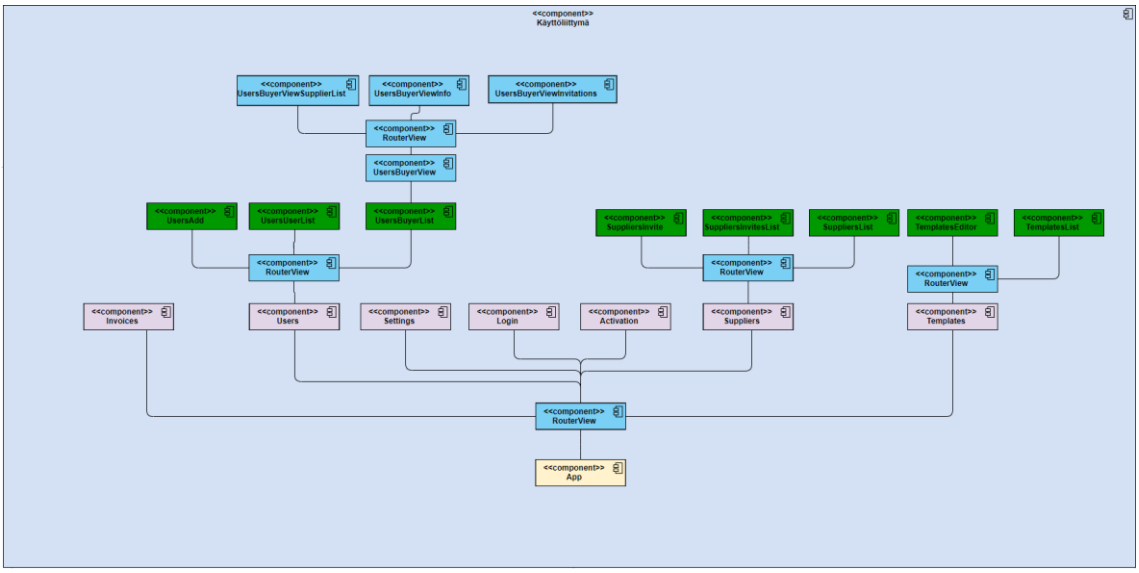


Kuva 7. Ylänavigointipalkki

Linkkien toiminnan yläpalkissa tarjoavat router-link-komponentit. Router-view-komponentti näyttää sisällään aina polulle kuuluvan komponentin. Toimittajarahoituksessa on hyödynnetty myös reitittimen polkujen children-ominaisuutta, jolloin sisäkkäisellä router-view'llä voi näyttää alanäkymän, jota havainnollistetaan kuvassa 8.



Kuva 8. Havainnollistus sisäkkäisistä näkymistä [19]



Kuva 9. Näkymätasot

Kuvassa 9 näkyy käyttöliittymän näkymä-komponentti-rakenne. Sovellus alkaa juuresta, josta löytyy App-komponentti. App-komponentin sisällä on ensimmäinen router-view-komponentti, joka toimii sovelluksen päätasona sekä sisällön näyttäjänä. Päätasolla ollaan URLin juuressa, esimerkiksi invoices-komponentti olisi ”/invoices”. Seuraavalle tasolle vievät käyttäjät, toimittajat ja pohjat -komponentit, joista löytyvät seuraavat router-view-komponentit. Näin saadaan alemman tason näkymiä. Kyseisissä komponenteissa on yksi tai useampi listanäkymä, joita ei voinut yhdistää samaan näkymään. Tämä alempi taso ilmenee URLissa ensimmäisen tason jatkeena, esimerkiksi suppliers-komponentin listanäkymä olisi ”/suppliers/list”. Yksittäisen ostajan näkymä, eli UsersBuyerView-komponentti on todellisuudessa ensimmäisellä tasolla, sillä se ei ole users-komponentin lapsi, vaan se on irrallaan kaikista. Sen URL on silti ”/users/buyers/:id”, sillä siihen pääsee siirtymään vain käyttäjistä löytyvän ostajalistan kautta.

Ohjeita sovelluksen käyttöön antaa näkymässä esiin painettava tekstiruutu, jonka sisältö vaihtuu näkymästä riippuen. Reitittimen kautta haettu näkymän nimi annetaan suoraan kääntäjä-funktiolle, joka hakee kyseiselle näkymälle kuuluvan aputekstin. Tarpeelliseksi tässä tuli v-html-direktiivi, joka tulkitsee stringin sisällä olevat HTML-merkinnät ja luo niistä elementit, sillä eri näkymät saattoivat tarvita esimerkiksi kuvaelementtejä.

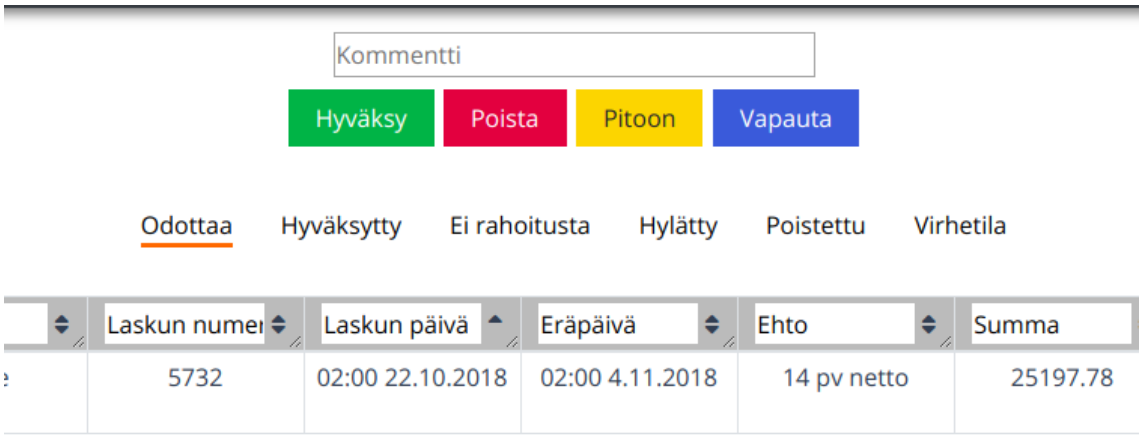
```
<div id="app">
        <app-nav></app-nav>
        <div class="container app-content" >
          <router-view></router-view>
          <help-dialog v-if="auth.isLoggedIn" :helpText="helpText"/>
          <notifications position="bottom right" :max="3"/>
        </div>
      <app-footer></app-footer>
  </div>
```

Esimerkkikoodi 30. Sovelluksen juuri. Tässä määritellyt komponentit ovat käytettävissä joka näkymässä, kuten ilmoitukset ja apu -dialogi.

Vuexin rooli toimittajarahoituksessa on pitää tallessa istunto- ja käyttäjätietoja, kuten asetuksia. Näkymiin haettava data tulee päivittää jokaisen siirtymän yhteydessä, joten niitä ei ole järkevää tallentaa Vuexin tilaan. Vuexin actionit toimivat hyvin rajapintakutsuna, sillä actionit ovat luonnostaan asynkronisia ja niitä pystyy kutsumaan kaikista komponenteista. Tämän takia Vuex toimii käyttöliittymän ja rajapinnan välikappaleena, kuten myös Vuex-kappaleen kuvassa 5 on esitetty.

5.2 Laskut-näkymä

Toimittajarahoitus pyörii laskujen ympärillä, joten luonnollisesti tärkein näkymä sovelluksessa on laskut-näkymä. Se mahdollistaa kaikkien osapuolien kanssakäymisen rahoitustoiminnassa. Järjestelmään tulleet laskut näytetään tässä näkymässä, ja ne siirtyvät tai ovat siirtymättä rahoitukseen valinnan mukaisesti. Rahoittaja näkee järjestelmän kaikki laskut ja niiden tilanteet. Toimittajat näkevät vain omat järjestelmään tulleet laskunsa, kun taas ostajat näkevät omat laskunsa kaikilta kutsumiltansa toimittajilta. Rahoittajalla, toimittajalla ja ostajalla on omat roolinsa laskuja koskevassa päätöksenteossa, ja kullakin on siis eri painikkeet esillä. Kuvassa 10 on rahoittajan näkymässä olevat painikkeet.



Kuva 10. Rahoittajan laskunäkymä

Laskun tilalla on ensisijainen merkitys, joten sen sijaan, että taulussa olisi ollut tilaa merkitsevä sarake, laskut on alun alkaen jaettu tiloihinsa ja tilojen välillä siirrytään taulukon yllä olevien otsakkeiden kautta, kuten kuvassa 10 näkyy. Taulukossa itsessään on sarakkeiden otsakkeissa syötekentät suodattamista varten.

Kuten aikaisemmin on määritelty, käyttäjän tulee pystyä vaihtamaan laskun tilaa. Sitä varten riveistä on tehty valittavia, sekä näkymään on lisätty toimintopalkki, jossa on painikkeita. Laskujen tiloja vaihtaakseen käyttäjä valitsee taulukosta rivejä ja painaa värikoodattua painiketta, jossa myös lukee tila, johon lasku siirretään. Käyttäjä voi halutessaan lisätä laskulle kommentin tilanvaihdoksen yhteydessä kommenttikenttää hyödyntämällä. Käyttäjä voi esimerkiksi kommentoida, miksi lasku hylättiin. Kuten aikaisemmin on mainittu, näkymä määräytyy käyttäjän roolin mukaan, joten tilat, joihin lasku on mahdollista vaihtaa, riippuvat myös siitä. Myös tilojen selaus on riippuvainen roolista. Esimerkiksi vain rahoittaja näkee poistetut laskut. Elementit luodaan kirjautuneen käyttäjän roolin perusteella, mutta myös rajapinnassa tarkistetaan oikeus toimiin. Laskujen tilavalinta on yhdistetty myös reitittimen hash-merkinnän avulla, jolloin sivun päivittämisenkin jälkeen pysytään tiettyjen laskujen näkymässä.

Taulukko on vue-tables-2-taulukkokomponentti. Tämän komponentin käyttö asetti hieman rajoituksia taulukon ominaisuuksille ja ulkoasulle, joiden kiertäminen vei aikaa.

Komponentti tarjosi kuitenkin paljon tehtyä työtä mahdollistamalla sivujen selauksen, rivien järjestämisen, suodattamisen sekä taulun tapahtumien kuuntelun. Komponentille annetaan options-objekti, jonka kautta sitä pystyy määrittelemään haluamakseen tiettyjen rajojen sisällä. Yksi korjattava epäkohta oli suodatuskenttien sijaitseminen omalla rivillänsä taulukon otsakerivin jälkeen. Suodatinkentät sai siirrettyä suoraan otsakkeisiin hyödyntämällä taulukon options-objektin otsakemäärittelyä. Tähän pystyi luomaan syötekentän antamalla renderöintifunktion. Taulukkoja ollessa useampi ja otsakkeita monia, oli järkevää tehdä yksi tehdasfunktio, jolla näitä otsakesuodattimia voi luoda. Esimerkkikoodissa 31 on laskutaulukon sovitus funktiolle, sillä taulu poikkeaa hieman muista sovelluksen tauluista. Koodissa on options-objektin sisälle määritelty taulun otsakkeeseen headerFilter-funktio. that viittaa Vue instanssiin ja $t viittaa Vuex i18n-kirjaston käännösfunktioon.

```
headings: {
// …
  paymentTerm: that.headerFilter(
               that.$t("invoices.paymentTerm"),
               "paymentTerm"
             ),
// …
```

Esimerkkikoodi 31. Options-objektin otsakemäärittely

Itse suodatuskentän tehdasfunktio on esimerkkikoodissa 32. Funktio luo syötekentän, johon on lisätty kuuntelijoita. Kuuntelijat on yhdistetty taulukon toimintaan siten, että enteriä painaessa taulukkoa suodatetaan syötekentän arvon mukaisesti. This.$refsin kautta käytetään taulukkokomponentin funktioita, joiden avulla lähetetään suodatuskutsu palvelimelle.

```
headerFilter: function(tString, columnName) {
      return function(h) {
        return h("input", {
          attrs: {
            placeholder: tString,
            class: "header-input"
          },
          on: {
            click: e => {
              e.stopPropagation();
            },
            keyup: e => {
              if (e.target.value === "") {
                this.filterColumn = "";
                this.filterValue = "";
                this.$refs.invoiceTable.getData();
              }
              if (e.code === "Enter") {
                let value = e.target.value;
                this.filterColumn = this.columnMap[columnName];
                this.filterValue = value;
                this.$refs.invoiceTable.getData();
                this.$refs.invoiceTable.setPage(1);
              }
            }
          }
        });
      };
    },
```

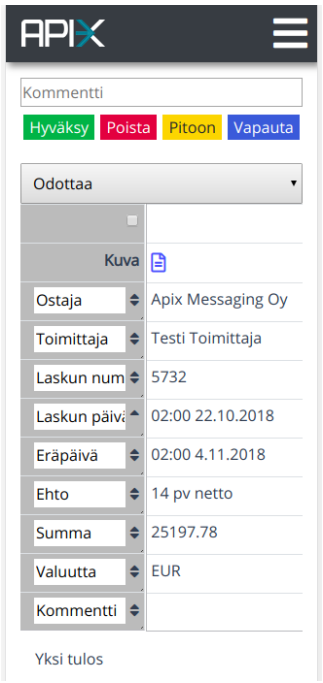
Esimerkkikoodi 32. Tehdasfunktio suodatinkentälle

Laskutaulukolle on valittu server-table-komponentti tavallisen asiakaspään taulukkokomponentin sijaan. Server-table on siirtänyt sivujen selaamisen ja rivien suodattamisen palvelimelle sen sijaan, että ne tehtäisiin asiakaspäässä. Tämä vaatii palvelinpään sovittamisen kutsuihin, joita taulukko lähettää. Taulukon tekemät pyynnöt ja niiden palautukset olivat muokattavissa, mikä helpotti sovitusta. Valinta oli selvä, sillä laskujen määrä tulee vain kasvamaan järjestelemässä, jolloin kaikkien laskujen kerralla hakeminen ja suodattaminen olisi kuormittavaa käyttöliittymälle. Tällöin voidaan varmistaa, ettei laskujen selaaminen hidastu suurienkaan määrien kanssa. Suodatuskutsun tuli ottaa huomioon aina valittu laskun tila sarakkeeseen kohdistetun hakutekstin lisäksi.

Palvelimen päässä selaaminen on toteutettu MySQL-tietokantakutsun limit-, offset- ja order by -määrittelyillä, jossa limitille annetaan palautettavien tietueiden määrä ja offsetille ohitettavien tietueiden määrä. Selauspyynnössä annetun sivun ja sen koon avulla parametrit voi laskea. Myös order by määrittelyä tulee käyttää, ja se on luonnollisesti taulukossa määritelty, eli minkä mukaan rivit järjestetään. Offsetin käyttö hidastaa hakuja, mitä suuremmaksi se käy, eli tässä tapauksessa kuinka pitkälle laskuja selataan.

Limitin ja offsetin yhdistämisen hitaus johtuu siitä, että offset hakee aina kaikki tietueet, mutta ei kuitenkaan palauta niitä. Tällöin voidaan kuvitella, että viimeistä sivua varten haetaan 5000 laskua ja näytetään niistä vain 10 viimeistä. Optimaalisempi tapa olisi käyttää tietueiden ID-numeroita ja aloittaa seuraava haku aina tietyn ID:n kohdalta (suurempi kuin haulla), mutta koska laskuja ei järjestetä ID:n mukaan, se ei ole mahdollista. Hyvällä suodatuksella voidaan kuitenkin varmistaa, ettei käyttäjän tarvitse siirtyä ensimmäisiä sivuja pidemmälle. [30; 31.]

Taulukkojen sovittaminen mobiilinäkymään osoittautui haastavaksi etenkin, kun sarakkeiden määrä kasvaa suureksi. Helppo ratkaisu olisi ollut lyödä taulukkoon overflow-määrittely, joka tekee siitä sivulle vieritettävän. Tämä ei kuitenkaan ollut tyydyttävä ratkaisu. Verkosta löytyi kuitenkin flip-table-määrittely, joka nimensä mukaisesti kääntää taulukon sivuttaiseksi, joka sopii hyvin puhelimen näytölle pyyhkäisyllä selattavaksi [32]. Toinen vaihtoehto olisi ollut myös No More Tables -määrittely, mutta se olisi vaatinut suodatinkenttien monistamisen ja sen, että laskun tila ja toimintopalkki olisivat seuranneet ruudulla. Flip-table-ulkoasu kuitenkin mahdollisti, että puhelimen näytöllä näkyy kaikki tarvittava, joten se oli selvä valinta. Otsakkeet suodatekenttineen ovat myös piilotettavissa, jos haluaa tehdä enemmän tilaa tietojen katsomiselle. Ratkaisun sovittaminen ei ollut kuitenkaan helppoa, sillä taulukolle oli jo valmiiksi määritelty edellä mainittu ”helppo” ratkaisu, ja taulukon HTML-rakenne oli epäsuotuisa. Laskun tilat on tässä myös siirretty pudotusvalikkoon. Kuvassa 11 on laskunäkymä mobiilinäkymällä, jossa taulukon sarakkeet on kasattu yhdeksi pylvääksi. Laskuja selataan pyyhkäisemällä sivulle.



Kuva 11. Laskunäkymä mobiililla

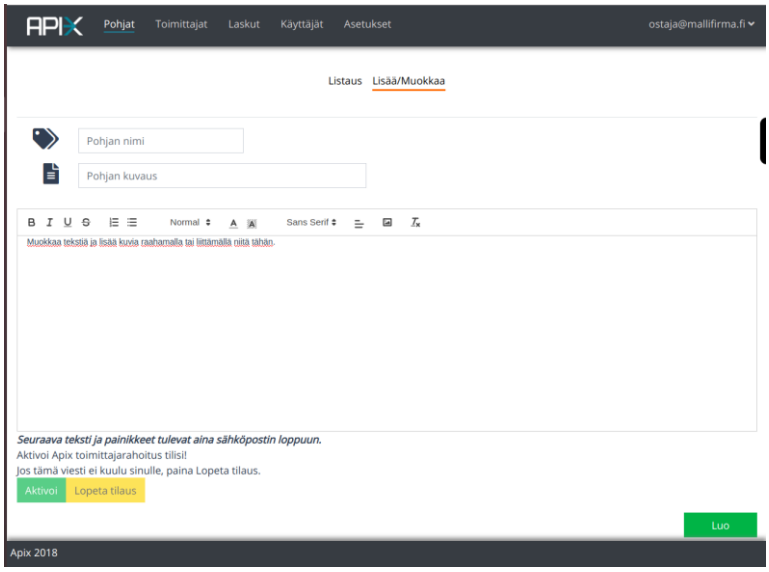
Käytettävyyteen on myös panostettu tekemällä mahdolliseksi liikkua taulukon riveillä tabulaattorilla. Liikkuminen heijastuu korostamalla aina se rivi, jolla liikutaan. Rivin voi valita tai poistaa valinnasta painamalla enteriä. Toteutus tehtiin yhdistämällä rivin sisällä olevan HTML-elementin tabindex-ominaisuutta ja CSS:n focus-inside-määrittelyä. Tabindexin avulla tavallinen div-elementti oli myös valittavissa tabulaattorilla ja siten siihen pystyi lisäämään enter-painikkeen kuuntelijan.

5.3 Pohjat-näkymä

Yksi sovelluksen tavoitteista oli mahdollistaa toimittajien automatisoitu lisääminen ostajien toimesta. Sen toteutus käyttöliittymässä koostuu kolmesta vain ostajalle kuuluvasta näkymästä: pohjat, toimittajat ja aktivointi. Toimittajakutsun ensimmäinen vaihe alkaa kutsupohjan luomisesta pohjat-näkymässä, jossa luodaan toimittajan sähköpostiin lähetettävälle kutsulle pohja.

Pohjat-näkymässä on alanäkyminä yksinkertainen listaus pohjille sekä muokkaamiseen tai uuden pohjan luomiseen tarkoitettu näkymä (kuva 12). Pohjat ovat yrityskohtaisia eli yhteisiä pääkäyttäjälle ja tämän alikäyttäjille. Muokkausnäkymässä pohjalle annetaan nimi, kuvaus ja sisältö. Sisällön pystyy luomaan tekstieditorilla. Kutsupohjien loppuun lisätään aina painikkeet, joilla toimittaja voi hyväksyä kutsun tai hylätä sen saadessaan kutsun sähköpostiinsa. Painikkeet ohjaavat tunnisteen kanssa sivulle, jossa rekisteröidään päätös.

Tekstieditori on toteutettu vue-quill-js-komponentilla, joka on JavaScript-pohjaisen QuillJS:n sovitus Vuelle. Editorista löytyvät tavallisimmat ominaisuudet, kuten fonttien säätäminen ja kuvien upotus tekstiin (kuva 12). Syöte tallennetaan HTML-merkkauksena. Komponenttiin voidaan kytkeytyä ja kuunnella sen tapahtumia.



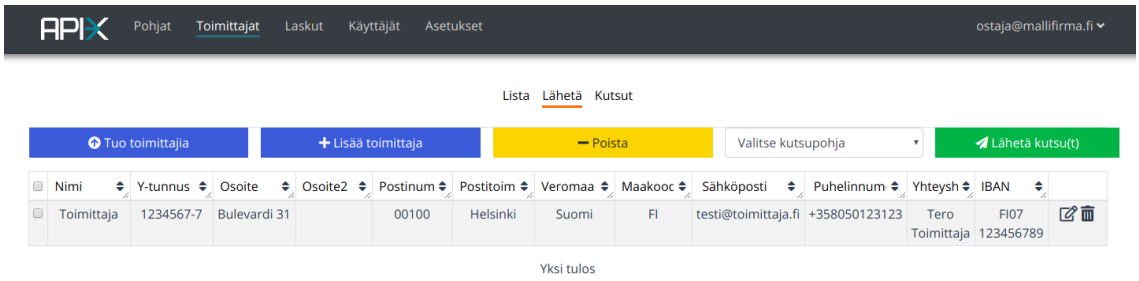
Kuva 12. Muokkaus-näkymä

5.4 Toimittajat-näkymä

Toinen vaihe toimittajien kutsumisessa toteutuu toimittajat-näkymässä (kuva 13). Täällä ostaja hallinnoi ja kutsuu lisää toimittajia. Toimittajat-näkymä sisältää lista-alanäkymän,

jossa ovat ostajan aktiiviset toimittajat. Lähetä-näkymässä lisätään kutsuttavaksi pyydettäviä toimittajia tietoineen ja lähetetään kutsuja. Näiden lisäksi on kutsut-näkymä, jossa näkee toimittajakutsujen tilan.

Näkymästä löytyvät toimittajahallintaan tarvittavat ominaisuudet. Suurien toimittajamäärien lisäämiseksi löytyy mahdollisuus ladata toimittajat listaan Excel-tiedostosta, mutta myös yksitellen lisääminen on mahdollista. Taulukosta voi myös valita useampia toimittajia kerralla poistettavaksi tai kutsuttavaksi. Kutsua lähettäessä tulee valita kutsupohja, jonka luomista käsiteltiin edellisessä luvussa. Toimittajia voi myös poistaa tai päivittää rivikohtaisesti rivien oikeassa laidassa olevien kuvapainikkeiden kautta.



Kuva 13. Toimittajakutsujen lähetysnäkymä

Toimittajataulukossa on jälleen käytössä vue-tables-2-taulukkokomponentti, tällä kertaa client-table, joka hoitaa suodatuksen ja tiedon järjestämisen asiakaspäässä. Taulukkokomponentti tarjoaa sarakkeiden muokkaamisen slot-menetelmällä, jossa sarakkeiden sisällön voi toteuttaa itse sen sijaan, että siinä näytettäisiin vain rivin tieto. Tämän avulla on toteutettu rivin lopussa olevat kuvakkeet, joiden avulla pystyy muokkaamaan riviä tai poistamaan sen.

```
<v-client-table :data="suppliers" :columns="columns"
:options="options">

//…

<template slot="edit" slot-scope="props">
  <div class="invites-action-cell"
       v-if="props.row.companyId === editing.companyId">
    <i @click="update" class="invites-action-cell-icon far fa-save"/>
    <i @click="editing = {index: null, companyId: null}"
    class="invites-action-cell-icon fas fa-ban"/>
  </div>

  <div class="invites-action-cell" v-else>
    <i @click="edit(props.index, props.row)"
     class="invites-action-cell-icon far     fa-edit"/>
     <i @click="deleteAction([props.row])"
      class="invites-action-cell-icon fas fa-trash-alt"></i>
  </div>
</template>

<v-client-table/>
```

Esimerkkikoodi 33. Toimittajataulukon edit-sarakkeen toteutus

Esimerkkikoodissa 33 on toteutus kuvakepainikkeiden sarakkeesta. Template-tagille annetaan slotissa sen sarakkeen nimi, joka korvataan, eli edit. Slot-scopessa määritellään, millä nimellä riviobjektiin pääsee käsiksi. Kuvakkeet on kääritty div-elementin sisään, joka näytetään ehtolauseella, mikä tarkistaa, ollaanko kyseistä riviä muokkaamassa. Jos riviä muokataan, näytetään ylempi div-elementti, jossa on kuvake tallentamista ja peruuttamista varten. Jälkimmäisessä div-elementissä on kuvakkeet poistamista ja rivin muokkaamista varten.

Samalla slot-menetelmällä on toteutettu sarakkeiden muokkaaminen. Valitun rivin kaikki tietoa sisältävät sarakkeet muutetaan syötekentiksi muokkaamista varten.

```
<template v-for="(field, key) in Object.keys(supplier)"
:slot="field" slot-scope="props">
     <div :key="key" class="edit-field">
          <input v-if="props.row.companyId === editing.companyId"
          v-model="supplier[field]"/>
          <div v-else>{{props.row[field]}}</div>
     </div>
</template>
```

Esimerkkikoodi 34. Rivimuokkaimen toteutus.

Esimerkkikoodissa 34 ehtolause muuttaa silmukan sisällä kaikki rivin sarakkeet syötekentiksi, jos rivi on valittu muokattavaksi. Muussa tapauksessa sarakkeen sisältö näytetään normaalisti. Tallenna painike rivillä lähettää rajapintaan muokatun tiedot toimittajasta.

Yksittäisen uuden toimittajan lisääminen tapahtuu painamalla ”lisää toimittaja” -painiketta. Painike avaa ikkunan, josta löytyvät kaikki täytettävät kentät. Intuitiivisempi ratkaisu olisi ollut taulukon riviltä löytyvä tyhjä rivi, jonka voi täyttää, mutta sen toteuttamiselle ei löytynyt ratkaisua client-tablen rajoitteista johtuen. Se olisi vaatinut ylimääräisen rivielementin, joka ei reagoi taulukon järjestämiseen, suodattamiseen tai sivun vaihtamiseen.

”Tuo toimittajia” -painike avaa ikkunan, jossa on painike tiedoston lisäämiselle. Vain Excel-tyyppiset tiedostot kelpaavat. Tiedoston valittua sen voi lähettää rajapintaan tulkittavaksi, jolloin sieltä palautuvat kantaan lisätyt toimittajat ja ilmoitus duplikaateista, jotka tarkistettiin y-tunnuksen avulla.

5.5 Aktivointinäkymä

Viimeinen vaihe toimittajan kutsumisessa on aktivointinäkymässä, johon toimittaja on saapunut toimittajakutsun kautta. Täällä toimittajasta löytyvät ostajan esitäyttämät tiedot, joita toimittaja voi korjata.

Aktivointinäkymä räätälöidään rahoittajan tarpeiden mukaan, jossa toimittaja solmii sopimuksen rahoittajan kanssa. Tässä aktivointinäkymä sisältää lomakkeen, johon toimittaja lisää yrityksensä tietoja. Mukaan tulee myös yhteyshenkilön tiedot skannatun henkilötodistuksen kanssa, jonka voi ladata pdf-muodossa. Yhteyshenkilö valitsee salasanan ja käyttäjätunnuksen, joka aktivoituu rahoittajan hyväksyessä sen.

Aktivointi näkymään pääsee vain yksilöidyllä linkillä, joka on lähetetty toimittajakutsun mukana. Tunnisteen avulla voidaan hakea rajapinnasta ostajan toimittajanäkymässä esitäyttämät tiedot, jolloin hakemuksen tekeminen on jouhevampaa toimittajalle.

Aktivointinäkymää varten luotiin syötekenttä-komponentti, jossa oli sisäänrakennettu validointi. Tällöin voidaan rajoittaa toisteisen validointilogiikan luomista pääkomponenttiin, eli käytännössä vältytään luomasta watcher jokaiselle syötekentän arvolle. Myös sama HTML-rakenne toistui jokaisen syötekentän ympärillä, kuten merkkaukset, jotka ilmaisevat syötekentän oikeellisuudesta. Syötekenttä saa propsien kautta selitetekstin, validointitilan, validaattorin ja arvon. Nämä näkyvät esimerkkikoodissa 35.

```
props: ['value', 'valid', 'label', 'validator',]
```

Esimerkkikoodi 35. Syötekentän propsit

```
<template>
    <div class="form-group col-lg">
        <label class="form-label">{{ label }}</label>
        <input
        maxlength="60"
        class="form-control form-control-sm"
        v-bind:value="value"
        v-on:input="$emit('input', $event.target.value)"
        v-bind:class="{'':valid === null,
                       'is-valid':valid === true,
                       'is-invalid':valid === false,
                       'no-validation':validator===null}">
        <div class="valid-feedback feedback-icon">
            <i class="fa fa-check"></i>
        </div>
        <div class="invalid-feedback feedback-icon">
            <i class="fa fa-times"></i>
        </div>
    </div>
</template>
```

Esimerkkikoodi 36. Syötekentän rakenne

Esimerkkikoodissa 36 syötekenttä kommunikoi isäntäkomponentin kanssa emit-funktion avulla. Syötekenttään tehdyt muutokset lähetetään isäntäkomponentille, jossa syötekentän todellinen tila sijaitsee ja joka tulee takaisin syötekomponentille propsin kautta. Tämä näkyy esimerkkikoodissa valuen ja inputin kohdalla, jossa ensimmäisessä emit-funktio ja jälkimmäisessä kytkös value-propsiin.

Syötteen oikeellisuuden tilaa merkitsee false-, true- tai null-arvo. Null tarkoittaa tyhjää tilaa, false virheellistä ja true on hyväksyttävä tila. Esimerkkikoodissa 36 näkyy, miten validoinnin tila on yhdistetty Bootstrap-kirjaston CSS-luokkaan v-bind:class-määrittelyllä,

jossa syötekenttä saa is-valid-luokan, jos valid-arvo on true. Tämän avulla esitetään kentän tila. Jos komponentti ei ole saanut validaattoria, eli syötteen tarkistajaa, sitä ei tarkisteta. Computed-arvo olisi ollut loogisempi vaihtoehto validoinnille, mutta koska se on reaktiivisempi, siihen ei voi vaikuttaa väliaikaisesti. Esimerkiksi tyhjää kenttää ei tarvitse ilmoittaa virheelliseksi, ennen kuin asiakas yrittää lähettää lomakkeen. Watcherin kanssa validointitilan voi asettaa virheelliseksi, joka muuttuu vasta sitten, kun kenttää aletaan taas täyttämään, sillä se reagoi vain syötteen muutokseen.

```
watch: {
        value: function(val) {
            if (this.validator) {
                this.$emit('validate', this.validator(val))
            }
        }
}
```

Esimerkkikoodi 37. Validointi-watcher

Esimerkkikoodissa 37 näkyy, miten validointitila päivitetään isäntäkomponenttiin lähettämällä validate-tapahtuman, jonka arvoksi tulee validoinnin arvo.

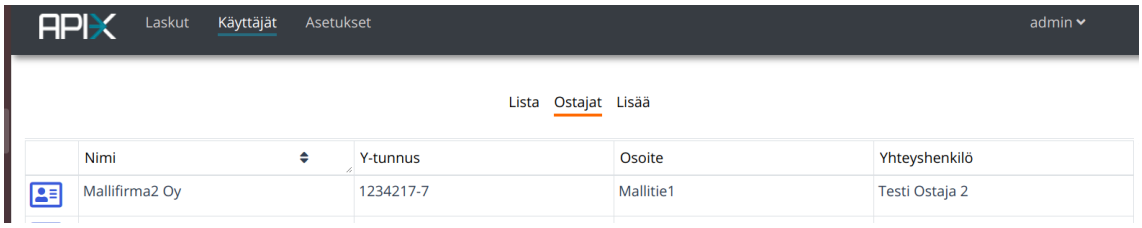
```
<activation-form-input
      :value="form.name"
      :label="$tlang(locale, 'activation.supplierName')"
      v-on:input="form.name = $event"
      :valid="form.validation.name"
      v-on:validate="form.validation.name = $event"
      :validator="simpleValidator"
      />
```

Esimerkkikoodi 38. Syötekenttäkomponentti lomakkeella

Esimerkkikoodissa 38 ollaan isäntäkomponentin, eli aktivointinäkymän tasolla. Syötekenttä on määritelty toimittajan nimi kentäksi ja saa validaattorikseen tavallisen syötteen pituuden tarkistajan. Label eli seliteteksti annetaan lomakkeella käytössä olevan kielen perusteella, joka poikkeaa asetusten kielimäärittelystä, sillä käyttäjä ei ole vielä kirjautunut eikä tällä ole asetuksia.

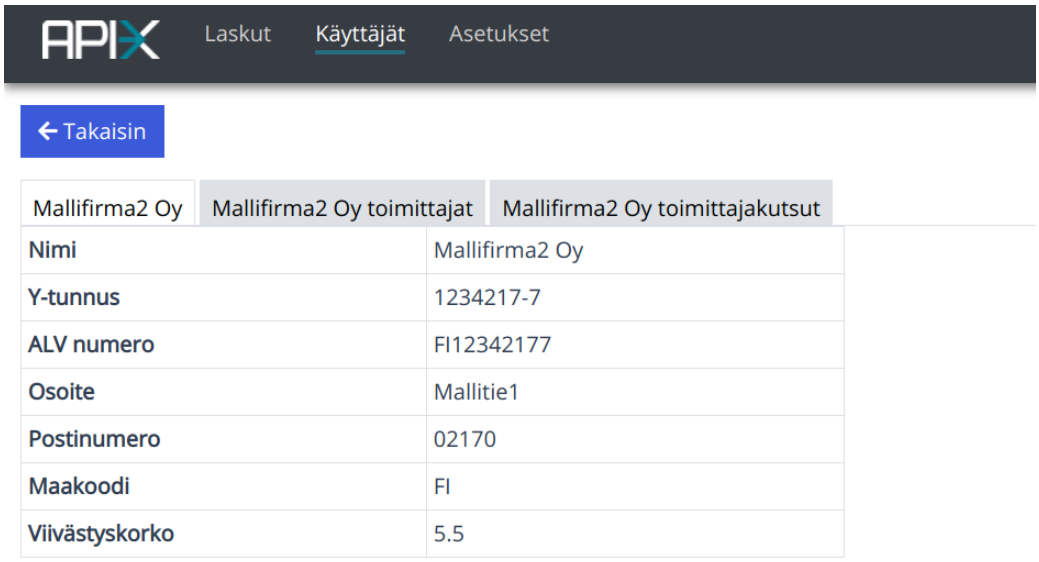
5.6 Käyttäjät-näkymä

Käyttäjänhallinnan työkaluna jokaisella pääkäyttäjällä toimii käyttäjät-näkymä. Täällä pääkäyttäjä voi tarkastella alikäyttäjiänsä ja luoda uusia. Alikäyttäjien tarkoitus on toimia laskunkäsittelijöinä. Kaikkien käyttäjien ollessa rahoittajan alaisia, pystyy rahoittaja tarkastelemaan kaikkia järjestelmän käyttäjiä. Rahoittajalla on erikseen listaus ostajista, josta pääsee ostajakohtaisesti näiden toimittajalistaukseen, jossa näkyvät myös toimittajakutsut. Täällä rahoittaja voi hyväksyä toimittajakutsuja. Rahoittaja näkee toimittajan perustietoja ja asettaa toimittajaa hyväksyessään sopimustunnuksen.



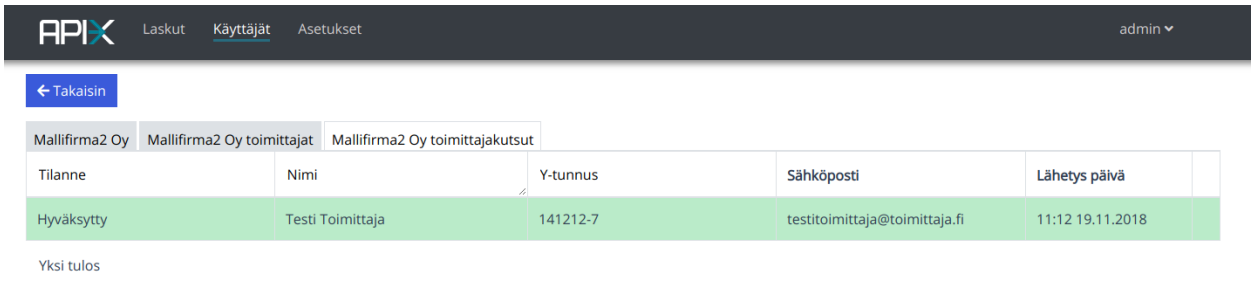
Kuva 14. Rahoittajan käyttäjänäkymä ostajista

Kuvassa 14 rahoittaja näkee listauksen ostajistaan ja pystyy tarkastelemaan ostajaa lähemmin painikkeesta rivin vasemmassa reunassa.



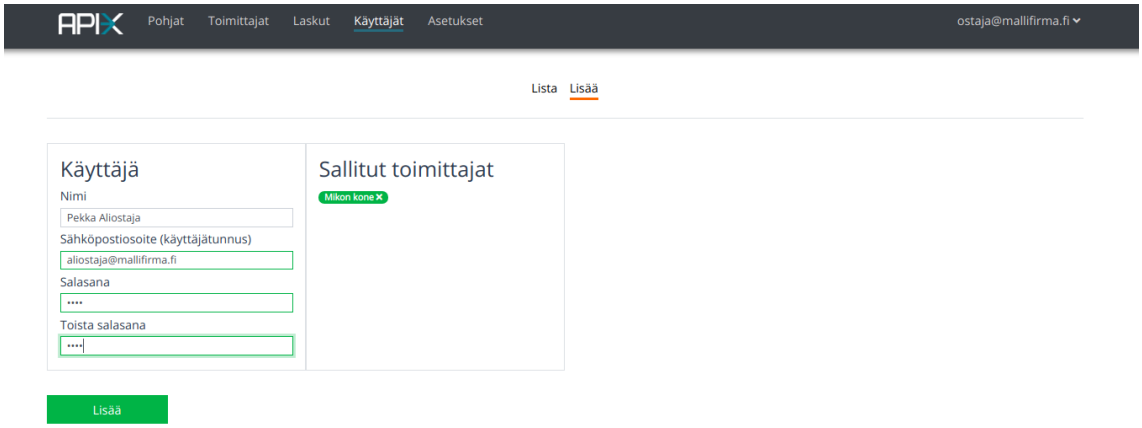
Kuva 15. Rahoittajan yksittäisen ostajan näkymä

Kuvassa 15 tarkastellaan ostajaa, ja välilehdistä voi tarkastella tämän toimittajia ja toimittajakutsuja. Tässä näkymässä on poikkeuksellisesti takaisin-painike, sillä se rikkoo kaksitasoisen navigoinnin. Kuvassa 16 on siirrytty toimittajakutsut-välilehdelle.



Kuva 16. Ostajan toimittajakutsut

Kuvassa 17 on ostajan perspektiivi käyttäjät-näkymästä. Ostajat näkevät käyttäjät-näkymässä vain alikäyttäjänsä, kun toimittajien hallinta on erikseen edellä läpikäydyssä toimittajat-näkymässä. Alikäyttäjää lisätessä ostaja voi päättää keiden toimittajien laskuja alikäyttäjä voi nähdä ja käsitellä.



Kuva 17. Ostajan alikäyttäjän lisäysnäkymä

5.7 Rajapinta

Palvelimen puolella pyörii Go:lla toteutettu REST-rajapinta. Rakenteellisesti kaikki resurssit sijaitsevat omissa pakkauksissansa ja jokaiselle resurssille on määritelty oma polku ja metodi, johon pyyntö tehdään. Joka polulle ja tähän kohdistettavalle HTTP-metodille on määritelty oma käsittelijäfunktio, handler. Metodit on jaettu siten, että GET-metodi on varattu tiedon noutamiseen, POST uuden luomiseen, PUT päivittämiseen ja DELETE poistamiseen. Useimmat resurssit ovat tunnistautumis-tokenin takana, joka annetaan HTTP-pyynnön *authorization headerissa*. Tokenin kautta saadaan selville käyttäjä ja se, mihin resursseihin hänellä on pääsy.

Pakkaukset ovat jaettu loogisiin kokonaisuuksiin, jotka vastaavat tietokannan rakennetta. Kokonaisuuksista tulee hieman sama jako kuin käyttöliittymässäkin. Pääasiassa jako on seuraava: laskut, toimittajat, kutsupohjat, kutsut ja käyttäjä. Jokaiselle pakkaukselle on määritelty oma server struct, jolta löytyy sille pakkaukselle kuuluvat handlerit

metodeina. Nämä handler-metodit yhdistetään reitittimen polkuihin. Handlerissa tavallisesti tarkistetaan käyttäjä ja onko hänellä oikeus resursseihin sekä onko pyyntö validia muotoa. Handler kutsuu sitten funktiota, joka tekee tietokantatoiminnot ja palauttaa vastauksen. Tämän jälkeen handler lähettää palautuksen palvelimelta asiakaspäähän. Pakkauksia on tämän lisäksi muun muassa autentikointia ja muuta apulogiikkaa varten. Nämä ovat rakenteeltaan erilaisia kuin edellä mainitut.

```
invoiceServer := invoices.Server{DB: db}
r.HandleFunc("/invoices", invoiceServer.HListInvoices).Methods("GET")

r.HandleFunc("/invoices/download/{uuid:[A-z0-9-]+}", invoiceServer.HGet-
InvoiceImage).Methods("GET")

r.HandleFunc("/invoices/{uuid:[A-z0-9-]+}", invoiceServer.HGetInvoice).Methods("GET")

r.HandleFunc("/invoices",invoiceServer.HProcessInvoice).Methods("PUT")
```

Esimerkkikoodi 39. Laskuihin liittyvien pyyntöjen käsittelijät

Esimerkkikoodissa yläpuolella näkyy polut laskujen hakemiseen ja käsittelemiseen. Ensin luodaan server struct, jolle annetaan tietokantayhteyttä varten konfiguraatio. HandleFunc kuuluu Mux-kirjaston reitittimelle. HandleFuncille annetaan polku, handler ja HTTP-metodi. Yksittäisen laskun ja tämän kuvan näkemistä varten pyyntöön tulee laskun tunniste. PUT-metodi on laskun tilan muuttamista varten ja pyyntöön lisätään JSONia, josta löytyy laskun tunniste ja siihen kohdistettavat muutokset. Listaukseen tulee useita hakuparametreja, jotka server-table-komponentti lisää pyyntöön automaattisesti.

```
var req InvoiceRequest
query := r.URL.Query()

req.Ascending = query.Get("ascending")
req.InvoiceStatus = query.Get("status")
req.FilterByColumn = query.Get("byColumn")
req.OrderBy = query.Get("orderBy")
finance, err := strconv.ParseBool(query.Get("financing"))

if err != nil {
   utils.HandleErrorToHTTPResponseWriter("json", w, err.Error(), http.StatusBadRequest)
   return
}
```

Esimerkkikoodi 40. GET-pyynnön käsittely laskujen listaus-handlerissa

Esimerkkikoodissa 40 näkyy osa laskujen listauksen handlerin koodista, joka ottaa GET-pyynnöstä talteen parametrit ja luo niistä pyyntö-structin lähetettäväksi eteenpäin. Kaikki parametrit tulevat string-tyyppisinä, joten booleanit ja int-tyypit tulee parsia stringeistä. Jos parsiminen ei onnistu, palautetaan virhe huonosta pyynnöstä. Kun pyynnöstä on

rakennettu struct, se tarkistetaan ja siirretään seuraavalle funktiolle, joka tekee tietokantapyynnön. Sitä ennen haetaan tietokannasta tokenin avulla käyttäjä, jolloin nähdään tämän rooli.

```
user, err := authentication.GetUserFromHTTPRequest(s.DB, r)
if err != nil {
    logrus.WithFields(logrus.Fields{
                            "reqID":    reqID,
                            "function": "HListInvoices",
                            "context":  "authentication.GetUserFromHTTPRequest",
                            "error":    err.Error(),
                        }).Error("")
    utils.HandleErrorToHTTPResponseWriter("json", w, err.Error(), http.StatusBadRequest)
    return
 }
```

Esimerkkikoodi 41. Käyttäjän tunnistus

Esimerkkikoodissa 41 on käytössä authentication-paketti, josta löytyvät kaikki käyttäjän tunnistukseen tarvittavat toiminnot. Tunnistusfunktiolle annetaan tietokantayhteys ja HTTP-pyyntö parametreina. Tunnistaminen palauttaa virheen epäonnistuessaan, jonka jälkeen tehdään lokimerkintä ja lähetetään virheviesti pyynnön tekijälle.

Alussa tietokantapyynnöt toteutettiin sqlx- ja sql -kirjastoilla, mutta SQL-lauseiden kirjoittaminen oli työlästä. SQL-lauseita joutui jatkuvasti tarkistamaan kehittimen ulkopuolella syntaksivirheiden varalta. Monet useamman taulun kyselyt osoittautuivat myös haasteellisiksi. Avuksi otettiin SQLBoiler-kirjasto. SQLBoiler ei ole yhtä rikas ominaisuuksiltaan kuin esimerkiksi Hibernate, mutta vähensi SQL:n kirjoittamista. SQLBoiler vaatii kuitenkin hieman konfigurointia, kuten työkalun ajamisen tietokantaa vasten, joka loi siten mallit relaatioista. Työkalua joutuu ajamaan aina, kun tekee tietokantaan muutoksia, sillä työkalun luomien mallien suoraa muokkausta ei suositella. Malleihin pystyy kuitenkin vaikuttamaan konfigurointitiedoston avulla, mikä on tärkeää, sillä käytännössä mallit ovat structeja metodeineen. Esimerkiksi tagit oli hyvä määritellä itse, jotta kommunikointi asiakaspään kanssa olisi sujuvaa. SQLBoiler tarjoaa structien metodien kautta tietokantahaut sekä luo suoraan structit lähetettäviksi asiakaspäähän, mikä vähentää kirjoitettavan koodin määrää.

6 Tulokset ja arviointi

Insinöörityö tarjosi oivalluksia ja erehdyksiä. Käyttöliittymästä tuli looginen ja selkeä osittain myös sen vuoksi, että toiminnallisuuksia oli vain vähän. Se olisi vaatinut kuitenkin alusta asti selkeämmän designin, sillä aikaa kului sommittelemiseen ja ulkoasujen koettamiseen, kuten esimerkiksi ylänavigaatiossa. Käyttöliittymässä kokeiltiin myös animaatioita, siinä luulossa, että niiden avulla sitä saisi siistimmäksi. Työstämisen kannalta turhautumista aiheutti taulukkokomponentti. Ajatus sen korvaamisesta kuitenkin torppasi siihen jo kulutetun ajan suuruus. Loppujen lopuksi taulukkokomponentista saatiin kuitenkin toimiva, joskin ratkaisemattomia ongelmia siitä jäi.

Myös rajapintasovelluksessa aiheutui uudelleenkirjoittamista SQLBoilerin käyttöönotossa. Sitä oli kuitenkin vaikeata arvioida, kuinka paljon hyötyä ja uudelleenkirjoittamista uuden kirjaston käyttöönotto aiheuttaisi. Jatkossa tämä hyöty kuitenkin lisääntyisi uusien ominaisuuksien mukana. Tyytyväisyyttä aiheutti laskuille toteutettu selaus palvelinpäähän, joka tässä työssä otettiin tutkittavaksi ja sen jälkeen käyttöön, sillä sille löytyy käyttökohteita muissakin projekteissa.

Lopputuloksena insinöörityölle saatiin toimiva sovellus. Myytäessä sovelluksesta saadaan taloudellista hyötyä yritykselle. Tämän lisäksi insinöörityö tuotti koodipohjaa ja komponentteja, joita on hyödynnetty jo muissa projekteissa. Työ opetti myös uusista teknologioista ja niiden sudenkuopista. Kehitystä tuli myös eri teknologioiden hyötyjen arvioimiseen, vaikkakin kokeilemalla se selviää edelleen parhaiten. Sovellus jää jatkuvan kehityksen alaiseksi, jolloin sitä parannellaan ja siihen lisätään ominaisuuksia tarpeiden mukaan.

7 Yhteenveto

Tässä insinöörityö raportissa kerrottiin uudesta myytävästä Apix Messaging Oy:n Toimittajarahoitussovelluksesta, joka mahdollistaa rahoituksen toimittajakeskeisestä näkökulmasta. Sovelluksen tulee mahdollistaa järjestelmään tulleiden laskujen esittämisen ja niitä koskevan päätöksenteon. Tavoitteena oli toteuttaa Toimittajarahoituksen käyttöliittymä ja rajapinta, joka hyödyntää valmista laskujenkäsittelyjärjestelmää.

Ensin käytiin läpi suunnittelua ja sitä, mikä on Toimittajarahoituksen toiminnallisuus sekä mitä teknologioita päätettiin sen toteuttamisessa käyttää. Seuraavaksi käytiin läpi käyttöliittymään valittuja teknologioita. Ensimmäiseksi aloitettiin käyttöliittymään käytettävän teknologian Vue.js:n perusteista ja sen käytöstä. Tämän jälkeen kerrottiin Golangin eli Googlen ohjelmointikielestä, jota käytettiin toteuttamaan Toimittajarahoituksen palvelinpään rajapinta.

Toteutus-luvussa esiteltiin työn tulosta kuvien ja esimerkkikoodien avulla sekä nousseita ongelmia. Valmistuneita näkymiä olivat muun muassa lasku- ja toimittaja-näkymät. Lopuksi kerrottiin rajapinnan toteutuksesta.

Insinöörityön lopputuloksena saatiin uusi myytävä tuote. Taloudellisen hyödyn lisäksi koodipohjaa ja komponentteja on hyödynnetty jo muissa projekteissa. Työn kautta on tullut myös uusia ideoita ja tapoja asioiden toteuttamiselle niin Vuella kuin Golangillakin. Sovellus jää jatkuvan kehityksen alaiseksi.

Lähteet

1 Käyttöpääoma ja käyttöpääoma-%. 2019. Verkkodokumentti. Alma Talent. <www.almatalent.fi/tietopalvelut/tunnuslukuopas/tehokkuus/kayttopaaoma-ja-kayttopaaoma-prosentti>. Luettu 12.4.2019.

2 Esa Stenberg. 19.12.2017. Laskut heti rahaksi: laskusaatavarahoitus. Verkkodokumentti. Finago. <blog.finago.com/fi/laskut-heti-rahaksi-laskusaatavarahoitus>. Luettu 12.4.2019.

3 Front-end Frameworks – Overview. 2018. Verkkodokumentti. Stateofjs. <2018.stateofjs.com/front-end-frameworks/overview/>. Luettu 8.3.2019.

4 Composing with Components. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/#Composing-with-Components>. Luettu 27.3.2019.

5 The Vue Instance. 2019. Verkkodokumentti. Vue.js <vuejs.org/v2/guide/instance.html>. Luettu 27.3.2019.

6 One-Way Data Flow. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/components-props.html#One-Way-Data-Flow>. Luettu 5.4.2019.

7 Computed Properties. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/computed.html#Computed-Properties>. Luettu 27.3.2019.

8 Watchers. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/computed.html#Watchers>. Luettu 3.4.2019.

9 Filters. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/filters.html>. Luettu 3.4.2019.

10 List Rendering. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/list.html>. Luettu 5.4.2019.

11 Conditional Rendering. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/conditional.html>. Luettu 5.4.2019.

12 Event Handling. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/events.html>. Luettu 5.4.2019.

13 Custom Events. 2019. Verkkodokumentti. Vue.js. <vuejs.org/v2/guide/components-custom-events.html>. Luettu 5.4.2019.

14 What is Vuex? 2019. Verkkodokumentti. Vuex. <vuex.vuejs.org/>. Luettu 17.2.2019.

15 Getters. 2019. Verkkodokumentti. Vuex. <vuex.vuejs.org/guide/getters.html>. Luettu 5.4.2019.

16 Mutations. 2019. Verkkodokumentti. Vuex. <vuex.vuejs.org/guide/mutations.html>. Luettu 17.2.2019.

17 Actions. 2019. Verkkodokumentti. Vuex. <vuex.vuejs.org/guide/actions.html>. Luettu 5.4.2019.

18 Form Handling. 2019. Verkkodokumentti. Vuex. <vuex.vuejs.org/guide/forms.html>. Luettu 5.4.2019.

19 Nested Routes. 2019. Verkkodokumentti. Vue Router. <router.vuejs.org/guide/essentials/nested-routes.html>. Luettu 5.4.2019.

20 Programmatic Navigation. 2019. Verkkodokumentti. Vue Router. <router.vuejs.org/guide/essentials/navigation.html>. Luettu 5.4.2019.

21 Navigation Guards. 2019. <https://router.vuejs.org/guide/advanced/navigation-guards.html>. Luettu 5.4.2019.

22 Getting Started. 2019. Verkkodokumentti. The Go Programming Language. <golang.org/doc/install>. Luettu 5.4.2019.

23 How to Write Go Code. 2019. Verkkodokumentti. The Go Programming Language. <golang.org/doc/code.html>. Luettu 5.4.2019.

24 The Go Programming Language Specification. 2019. Verkkodokumentti. The Go Programming Language. <golang.org/ref/spec>. Luettu 5.4.2019.

25 Nils in Go. 2019. Verkkodokumentti. Go101. <go101.org/article/nil.html>. Luettu 6.4.2019.

26 Structs and Interfaces. 2019. Verkkodokumentti. Golang book. <www.golang-book.com/books/intro/9>. Luettu 6.4.2019.

27 Go maps in action. 6.2.2013. Verkkodokumentti. The Go Blog. <blog.golang.org/go-maps-in-action>. Luettu 7.4.2019.

28 Go – Functions. 2019. Verkkodokumentti. Tutorialspoint. <www.tutorialspoint.com/go/go_functions.htm>. Luettu 7.4.2019.

29 Remote packages. 2019. The Go Programming Language. Verkkodokumentti. <golang.org/doc/code.html#remote>. Luettu 7.4.2019.

30 Robert Eisele. 24.10.2011. Optimized Pagination using MySQL. Verkkodokumentti. <www.xarg.org/2011/10/optimized-pagination-using-mysql>. Luettu 22.2.2019.

31 Tomer Shay. 23.7.2017. Faster Pagination in Mysql – Why Order By With Limit and Offset is Slow? Verkkodokumentti. <www.eversql.com/faster-pagination-in-mysql-why-order-by-with-limit-and-offset-is-slow>. Luettu 22.2.2019.

32 Simon Elvery. Responsive Tables Demo. Verkkodokumentti. <elvery.net/demo/responsive-tables>. Luettu 22.2.2019.