Haaga-Helia
ammattikorkeakoulu Oy

Päiväkirjaopinnäytetyö etäterapiapalvelun kehittäjänä

Mikko Maja

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2020

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

21.05.2020

Tekijä(t)
Mikko Maja

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Päiväkirjaopinnäytetyö etäterapiapalvelun kehittäjänä

Sivu- ja liitesivumäärä
62 + 0

Opinnäytetyön otsikko englanniksi
Work diary thesis of developing a remote therapy service

Tässä päiväkirjaopinnäytetyössä kerron työstäni Navisec Health -etäterapiapalvelun kehittäjänä. Raportoin päiväkohtaisesti päivän tehtäviäni kolme kertaa viikossa kymmenen viikon ajan. Viikon lopuksi arvioin ja analysoin omaa suoriutumistani työssä.

Käytän työssäni pääasiassa JavaScriptin React-kirjastoa ja Pythonin Django-sovelluskehystä. Työhön liittyy myös muita teknologioita, kuten konttiteknologia Docker ja versionhallintajärjestelmä Git.

Yksittäisiä kehityskohteita oli seurantajakson aikana vaikea havaita, sillä olin toiminut työssäni ennen opinnäytetyön tekoa jo yli puoli vuotta. Kehitystä tapahtui kyllä jatkuvasti, mutta sitä saattoi olla viikkotasolla vaikeaa huomata. Pikkuhiljaa opin kuitenkin tunnistamaan omaa kehitystäni niin itsenäisessä työskentelyssä, kuin ongelmanratkaisussakin. Päiväkirjaraportoinnin myötä opin myös enemmän käyttämistäni teknologioista.

Asiasanat
Ohjelmistokehitys, ohjelmointi, JavaScript, Python

Sisällys

1 Johdanto 1
1.1 Finnish Net Solutions Oy 1
1.2 Navisec Health 1
1.3 Tietoperusta ja vaadittava osaaminen 2
1.4 Käsitteitä 2
2 Lähtötilanteen kuvaus 4
2.1 Oman nykyisen työn analyysi 4
2.2 Sidosryhmät työpaikalla 6
2.3 Vuorovaikutustaidot työpaikalla 7
3 Päiväkirjaraportointi 9
3.1 Seurantaviikko 1 9
3.2 Seurantaviikko 2 17
3.3 Seurantaviikko 3 21
3.4 Seurantaviikko 4 25
3.5 Seurantaviikko 5 29
3.6 Seurantaviikko 6 35
3.7 Seurantaviikko 7 40
3.8 Seurantaviikko 8 45
3.9 Seurantaviikko 9 51
3.10 Seurantaviikko 10 56
4 Pohdinta ja päätelmät 60
Lähteet 61

1 Johdanto

Päiväkirjamuotoisessa opinnäytetyössä raportoin ja analysoin päivittäistä työskentelyäni Navisec Health -etäterapiapalvelun kehittäjänä. Työ on aloitettu 19.02.2020 ja se on tarkoitus saada valmiiksi 19.05.2020 mennessä.

Opinnäytetyön kirjoittamisen aikana työskentelen Finnish Net Solutions -nimisessä ohjelmistotalossa osa-aikaisena työntekijänä (15 - 37,5h/vko) Web Developerina. Viittaan yritykseen tekstissä lyhenteellä FNS.

1.1 Finnish Net Solutions Oy

Finnish Net Solutions on 2001 perustettu ohjelmistotalo, joka kehittää valmiita tuotteita eläinterveydenhuoltoon sekä terapia-alan ammattilaisille. FNS:lla on toimistoja neljässä eri kaupungissa, ja työntekijöitä on yhteensä reilu sata. Itse työskentelen Espoon toimistolla, jonka sisällä toimii kolme tiimiä, joista kaksi kehittävät FNS:n päätuotteita, Diariumia ja Provet Cloudia. Kolmas tiimi, johon itse kuulun, on nimeltään Advanced Technologies, jossa kehitetään useita talon ulkopuolisia tilausprojekteja ja paria talon sisäistä palvelua. Omat työtehtäväni liittyvät pääasiassa yrityksen omistaman Navisec Health -etäterapiapalvelun kehitykseen.

Advanced Technologies -kehittäjätiimissä toimii tällä hetkellä itseni lisäksi kuusi henkilöä, joista kolme ovat opiskelijoita vähäisellä työkokemuksella. Heidän lisäkseen tiimistä löytyy kaksi kokeneempaa *Django*-kehittäjää ja tiiminvetäjä. Navisec Healthin parissa työskentelee tällä hetkellä pääsääntöisesti tiimin opiskelijat ja tiiminvetäjä, mutta kaikki kehittäjät ovat olleet mukana jossain vaiheessa projektia ja tuntevat sen varsin hyvin.

1.2 Navisec Health

Navisec Health on terveydenhuollon ammattilaisille ja heidän asiakkailleen tarkoitettu turvallinen etäterapiapalvelu. Palvelussa eri alojen terapeutit voivat kommunikoida turvallisesti asiakkaidensa kanssa ja lisätä palveluun erilaisia kursseja, lomakkeita ja aikataulluttaa niihin liittyviä toimintoja omille asiakkailleen. Jokainen asiakas saa itselleen kevyesti räätälöidyn version Navisec Healthista ja maksaa kuukausimaksua, joka määräytyy halluttujen ominaisuuksien mukaan. Palvelua on kehitetty vasta kahden vuoden ajan, joten uusia ominaisuuksia kehitetään jatkuvasti samalla kun nykyisiä ominaisuuksia parannetaan toimimaan usealla eri alan asiakkaalla.

Selaimessa toimivan palvelun backend on toteutettu Djangolla (Python) ja frontend *Reactilla* (JavaScript). Projektissa ei käytetä Scrumia tai muuta sen kaltaista projektinhallinnan viitekehystä, mutta lyhyissä viikkopalavereissa käydään läpi missä mennään ja mitä työlistalla odottaa seuraavana. Pääasiallinen kommunikointi käydään yrityksen Slack-kanavilla. Projektin koodikanta sijaitsee *GitLabissa*, joka toimii hyvin pitkälti kuten GitHub, mutta pyörii yrityksen omalla palvelimella, jotta sinne ei pääsisi käsiksi ulkopuolelta. Projektiin liittyvät dokumentit, tehtävätaulut ja roadmap löytyvät tehtävienhallintaohjelmisto *Jirasta*.

1.3 Tietoperusta ja vaadittava osaaminen

Navisec Healthia kehitetään Pythonilla ja JavaScriptilla. Palvelun backend on rakennettu Pythonin Django-sovelluskehyksellä ja frontend JavaScriptin React-kirjastolla. Djangon ja Reactin virallisten dokumentaatioiden ja lukuisten erilaisten blogikirjoitusten ja artikkeleiden lisäksi olen valinnut oppimisen tueksi *Eloquent JavaScript* ja *React Pro 16* -kirjat. Eloquent JavaScript käsittelee ohjelmointia yleisesti, JavaScript -kieltä ja sen käyttöä web-selaimissa. Kirja on varsin suosittu, ja siitä julkaistiin hiljattain jo kolmas versio. Kirja on luettavissa ilmaiseksi internetissä ja kaikki siitä löytyvät koodiesimerkit ovat kätevästi muokattavissa ja ajettavissa suoraan selaimessa. React Pro 16 on opaskirja modernin Reactin kehittäjille. Se ei sisällä Reactin aivan uusimpia ominaisuuksia kuten Hookeja, mutta näitä ei käytetä Navisec Healthissakaan, joten kirja on juuri sopivan ajankohtainen omaan tarkoitukseeni.

Pythonin ja JavaScriptin lisäksi työssäni on eduksi Docker -osaaminen. Palvelun kehitysympäristö pyörii Docker -kontissa, joten ainakin Dockerin peruskomentojen hallinta on osa jokaista työpäivää. Dockerin lisäksi Git -versionhallintajärjestelmän perustaidot tulee olla kohtuullisen hyvin hallussa.

1.4 Käsitteitä

Back-end	Sovelluksen taustalla pyörivä palvelinpuoli.
Front-end	Sovelluksen loppukäyttäjälle (tässä tapauksessa selaimessa) näkyvä osuus.
React	Komponenttipohjainen JavaScript-kirjasto web-käyttöliittymien tekoon.

Django	Pythoniin pohjautuva sovelluskehys web-sovellusten luomiseen.
Scrum	Projektinhallinnan viitekehtys, jota usein käytetään ketterässä ohjelmistokehityksessä.
ES6	JavaScriptin käyttämän ECMAScript -standardin kuudes versio.
Slack	Organisaatioiden sisäiseen viestintään suunnattu pikaviestintäsovellus.
Jira	Tehtävienhallintaohjelmisto, jonne yksittäiset työtehtävät yleensä kirjataan.
API	Ohjelmistorajapinta, joka mahdollistaa eri ohjelmien kommunikoinnin keskenään.
Bugi	Virhe ohjelmistokoodissa.
Duplikaattikoodi	Pätkä koodia, joka toistuu samanlaisena useassa eri paikassa
Docker	Avoimen alustan sovellus, jolla voidaan pakata sovellus virtuaaliseen pakettiin, eli konttiin.
CSS	Cascading Style Sheets, verkkodokumenttien ulkoasua määrittelevä kieli.
Git	Komentorivillä toimiva hajautettu versionhallintajärjestelmä.
Gitlab	Gitlab on avoimen lähdekoodin versionhallintapalvelu Git-versionhallinnalle (Kaukoharju, A. 2017)

2 Lähtötilanteen kuvaus

2.1 Oman nykyisen työn analyysi

Työtehtäväni ovat pääasiassa kehityskeskeisiä. Tavallisimmat työtehtävät koostuvat Navisec Healthin uusien ominaisuuksien kehittämisestä, olemassa olevien ominaisuuksien parantamisesta ja bugikorjauksista.

Tarve uusille ominaisuuksille syntyy yleensä asiakkaan tarpeesta. Navisec Healthin tuoteomistaja arvioi, onko asiakkaan haluama ominaisuus järkevää toteuttaa, ja onko ominaisuus sellainen, että se on tarpeellinen kaikille Navisec Healthia käyttäville yrityksille. Haluttu ominaisuus esitellään yleensä kehitystiimille viikkopalaverin yhteydessä, missä pohditaan yhdessä tiiminvetäjän, tuoteomistajan ja tuoteasiantuntijan kanssa uuden ominaisuuden toteutusta ja aikataulua. Palaverin jälkeen uusi ominaisuus pilkotaan Jiran tehtävätaululle pieniksi kokonaisuuksiksi ja samalla määrätään kuka tai ketkä uutta ominaisuutta lähtevät kehittämään.

Tarve olemassa olevien ominaisuuksien parantamiselle tulee usein asiakkaalta, mutta voi myös herätä vaikkapa bugikorjausten yhteydessä tai päivittäisen kehityksen puitteissa. Parannusehdotukset ilmoitetaan yleensä projektin Slack -kanavalla tai viikkopalaverissa. Mikäli ehdotuksia päätetään ruveta kehittämään, luodaan niistä tehtävät Jiraan ja määrätään kuka tai ketkä muutokset toteuttavat. Pienemmille parannuksille ei yleensä ole määritelty aikataulua, vaan kehittäjät saavat sovittaa ne omiin aikatauluihinsa vapaasti.

Bugeja ja virheitä monitoroidaan *Sentry* -nimisellä palvelulla. Jos asiakas kohtaa virheen tai bugin, jota ei käsitellä koodissa, Sentry lähettää ilmoituksen sekä sille varatulle Slack-kanavalle että kehittäjien sähköposteihin. Ilmoitukset tarkistetaan mahdollisimman pian niiden saapumisesta ja niistä selvitetään, kuinka kriittinen virhe on kyseessä. Ilmoituksista luodaan tehtävä Jiraan ja mikäli kyseessä on kiireellinen tehtävä, määritetään sille yleensä korjaaja välittömästi.

Bugikorjausten ja vanhojen ominaisuuksien kehittämisen työnkulut ovat kehittäjien kannalta varsin samanlaiset. Käyn läpi Jiraan luodut tehtävät kuvauksineen, mahdolliset aiheeseen liittyvät Slack-keskustelut ja/tai bugiraportit. Tämän jälkeen käyn läpi koodikantaa ja hahmottelen joko päässäni tai paperille, mitkä komponentit vaativat muutoksia ja miten muutokset olisivat järkevintä toteuttaa. Jos olen epävarma jostain, pyydän mielipiteitä muilta kehittäjiltä. Pyrin myös tunnistamaan etukäteen, löytyykö järjestelmästä vastaavia

toiminnallisuuksia tai valmiita komponentteja, joita voisin hyödyntää, jotta projektin koodi pysyy johdonmukaisena eikä duplikaattikoodia pääse syntymään.

Uutta ominaisuutta kehitettäessä suunnitteluvaihe on yleensä merkittävästi laajempi. Uudet ominaisuudet todennäköisesti vaativat uusia tietokantatauluja, komponentteja ja mahdollisesti kolmannen osapuolen luomia moduuleja. Tässä vaiheessa on tärkeää hahmottaa, miten järjestelmän olemassa olevat komponentit liittyvät toisiinsa ja onko jonkin tarvittavan komponentin luonti järkevää toteuttaa alusta asti itse vai onko olemassa valmiimpi ratkaisu olemassa olevan komponentin tai ulkopuolisen moduulin muodossa.

Itse kehitystyössä tukeudun paljon olemassa olevaan koodiin ja tiedonhakuun internetistä. Projektilla ei oman harjoittelujaksoni aikana ollut lainkaan dokumentaatiota, joten alkuun käytin paljon aikaa koko koodikannan tutkimiseen ja keskityin tekemään pienempiä korjauksia ja muutoksia, joiden kautta pääsin pikkuhiljaa sisään kokonaisuuteen. Sittemmin dokumentaatiota on luotu ja tehtävien suorittaminen on viikko viikolta ollut helpompaa.

Osaamistasoltani koen tällä hetkellä olevani aloittelevan toimijan ja taitavan suoriutujan välimaastossa. En ollut ennen nykyistä työtäni koodannut Djangolla mitään, joten selviytyäkseni tehtävistäni olen joutunut lukemaan paljon dokumentaatiota ja kysymään neuvoja kokeneemmilta kehittäjiltä. Nykyisin osaan Djangon perusteet varsin hyvin, mutta tukeudun olemassa olevaan koodiin ja kyselen muilta kehittäjiltä melko usein vinkkejä, jotta oppisin asioita, joita en dokumentaatiosta ole osannut välttämättä etsiä. Yksi tärkeimmistä tavoitteistani onkin kirjoittamani koodin pysyminen siistinä ja helposti ylläpidettävänä. Testaus tuli itselleni myös kokonaan uutena asiana. Aluksi testien ajamiset unohtuivat pieniä muutoksia tehdessä useinkin. Vasta kun pääsin kehittämään kokonaan uutta ominaisuutta itsenäisesti, tajusin kunnolla testauksen hyödyllisyyden ja sitä myötä olen oppinut kirjoittamaan parempia testejä aikaisemmassa vaiheessa kehitystä.

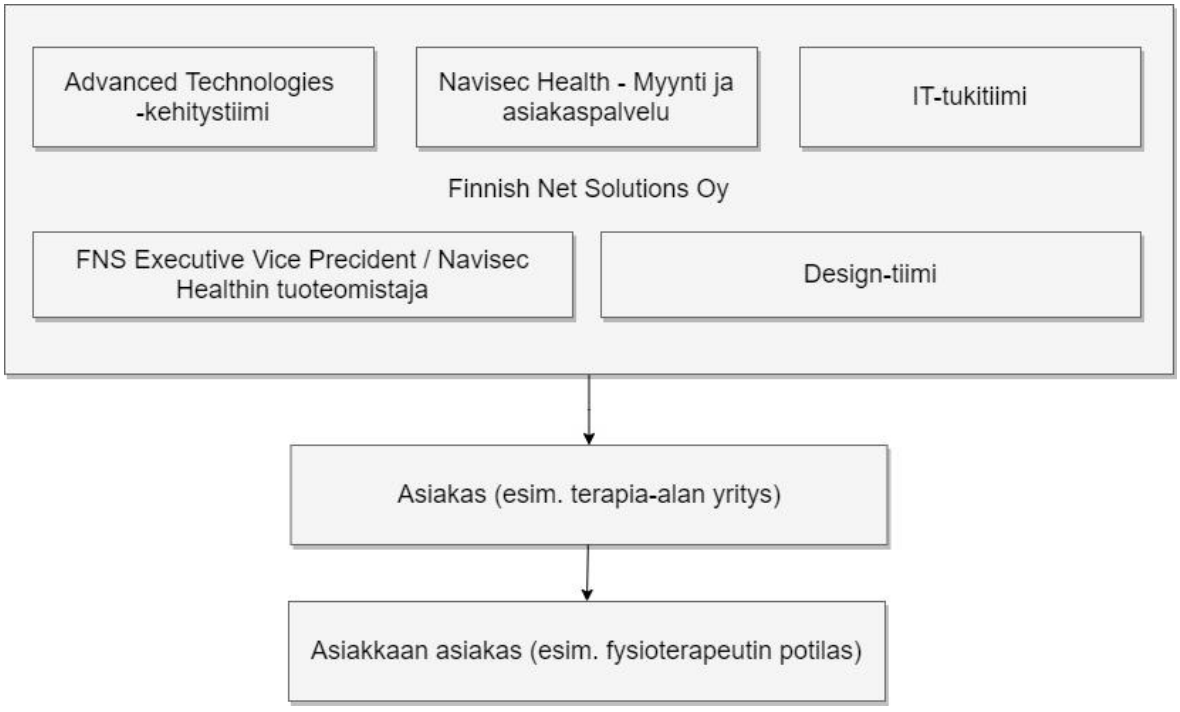
Frontend-puolen tehtäviä olen pystynyt hoitamaan pääasiassa itsenäisesti, mutta kahden aikaisemman React -kehittäjän erilaiset koodaustyylit ja paikoittain vaikeasti luettavat komponentit ovat hidastaneet tehtävien valmistumista. Olen saanut melko paljon hyvää palautetta tekemistäni lisäyksistä ja korjauksista, mutta olen itse usein kokenut, että tehtävään on kulunut liian kauan. Tiimissämme ei kuitenkaan ole tällä hetkellä yhtään kokeneempaa React -kehittäjää, joten ongelmanratkonta on perustunut pitkälti tiedonhakuun ja muiden harjoittelijoiden kanssa pohtimiseen. Nykyisin kykenen kuitenkin luomaan kokonaan uusiakin komponentteja itsenäisesti ja pitämään huolen, että ne ovat helposti luettavia ja järkevän kokoisia.

Nykyinen työni on käytännössä ensikosketukseni ohjelmistoalalla työskentelyyn. Olen noin kymmenen vuotta aikaisemmin tehnyt työtä HTML:n ja CSS:n parissa, mutta en koe siitä olleen hyötyä nykyistä työtäni ajatellen. Harppaus opiskeluaikana ja omalla ajalla tehdyistä projekteista varsinaiseen työpaikkaan oli varsin suuri. Harjoittelujaksooni ei sisältynyt mitään varsinaista koulutusta, joten oppiminen oli pitkälti omalla vastuullani. Kehitykseni harjoitusjakson aikana tuntui välillä todella hitaalta, eikä asiaa helpottanut samaan aikaan tiimissäni aloittaneen toisen harjoittelijan työn nopea sisäistäminen.

Parhaiten huomasin kehitykseni, kun tiimissämme aloitti syksyllä uusi opiskelija, joka korvasi suhteellisen kokeneen React-kehittäjän. Autoin hänet alkuun Navisec Healthin kanssa ja jaoin luomiani skriptejä muun muassa Django-testien ajoa helpottamaan. Huomasin tuntevani Navisec Healthin rakenteen ja toiminnot varsin hyvin ja olenkin pystynyt ehdottamaan useita parannuksia, jotka on otettu ilolla vastaan.

2.2 Sidosryhmät työpaikalla

Pienen, vielä kehityskaarensa alkupäässä olevan, tuotteen sidosryhmät ovat suurimmaksi osaksi sisäisiä. Sisäisiin sidosryhmiin kuuluvat Advanced Technologies -kehitystiimi, johon itse kuulun, Navisec Healthin tuoteomistaja ja tuoteasiantuntija sekä myyjät, jotka toimivat myös asiakaspalvelijoina, lisäksi yrityksen IT -tuki- ja design -tiimit.



Kuva 1. Sidosryhmät ohjelmistokehittäjän näkökulmasta

Pienessä seitsemän hengen Advanced Technologies -kehitystiimissämme kaikki eivät tällä hetkellä aktiivisesti osallistu Navisec Healthin kehitykseen, mutta tarvittaessa voin kysyä keneltä tahansa apua tai mielipidettä siihen liittyen.

Yrityksen toinen perustaja ja Navisec Healthin tuoteomistaja tekevät tuotetta koskevat kriittiset päätökset ja ylläpitävät roadmapia ominaisuuksista, joita tuotteeseen on suunniteltu lisättäväksi. Hän osallistuu aktiivisesti viikkopalavereihin ja pitää tiimin ajan tasalla aikatauluista ja asiakkaan tarpeista.

Navisec Healthin tuoteasiantuntija on tuoteomistajan ohella tärkeä linkki yrityksen ja asiakkaiden välillä. Hän on aktiivisesti yhteydessä asiakkaisiin, kuuntelee heidän toiveitaan uusista ominaisuuksista ja parannuksista, ja tiedottaa kehitettävistä ominaisuuksista ja niiden aikatauluista. Hän myös etsii aktiivisesti bugeja Navisec Healthin testiversiosta ja raportoi niitä Slack-kanavallemme. Apunaan hänellä on pieni myynti- ja asiakaspalvelutiimi, jotka yrittävät löytää tuotteelle uusia asiakkaita ja silloin tällöin raportoi teknisistä ongelmista kehitystiimille.

Yrityksen IT-tukitiimi vastaa Navisec Healthin palvelimista ja niiden tietoturvasta. Designtiimi taas tekee pyydettäessä ulkoasusuunnitelmia uusille ominaisuuksille.

Kehittäjän kannalta ainoat ulkoiset sidosryhmät ovat asiakkaat (esim. terapia-alan yritys) ja asiakkaiden asiakkaat (esim. terapeutin potilaat). Navisec Healthin käyttö perustuu terapeutin ja potilaan väliseen interaktioon, joten kaikissa ominaisuuksissa pitää huomioida molempien tarpeet.

2.3 Vuorovaikutustaidot työpaikalla

Päivittäinen vuorovaikutus tapahtuu yleensä kehitystiimin kesken. Avokonttorilla suurin osa tiimistämme tykkää työskennellä omassa rauhassaan vastamelukuulokkeet päässä ja kommunikoida asiansa Slackin kautta. Etenkin vähemmän kokeneiden tiimikavereiden kesken käymme paljon keskustelua kasvokkain ja autamme toisiamme tarvittaessa. Emme kommunikoi asiakkaidemme kanssa suoraan käytännössä koskaan; tuoteomistaja, tuoteasiantuntija ja myyjät hoitavat asiakaskommunikaation ja välittävät tarvittaessa viestejä yleensä Slack-kanavallemme. Me vuorostamme ilmoitamme esimerkiksi bugikorjauksen tuoteasiantuntijalle, joka tiedottaa siitä asiakkaalle. Talon sisäisten sidosryhmien kanssa kommunikoimme yleensä palavereiden yhteydessä tai Slackin kautta.

Olen pitänyt vuorovaikutustilanteita työssäni todella helppoina. Kaikkien sisäisten sidosryhmien jäsenten kanssa kommunikointi on sujuvaa ja tulen hyvin kaikkien kanssa toimeen. Alkuun tuntui oudolta laittaa samassa pöytäryhmässä istuville työntekijöille viestiä Slackin kautta, mutta myöhemmin olen huomannut itsekin, että omaan työhön on helpompi keskittyä, kun jokaiseen viestiin ei tarvitse välttämättä reagoida välittömästi.

3 Päiväkirjaraportointi

3.1 Seurantaviikko 1

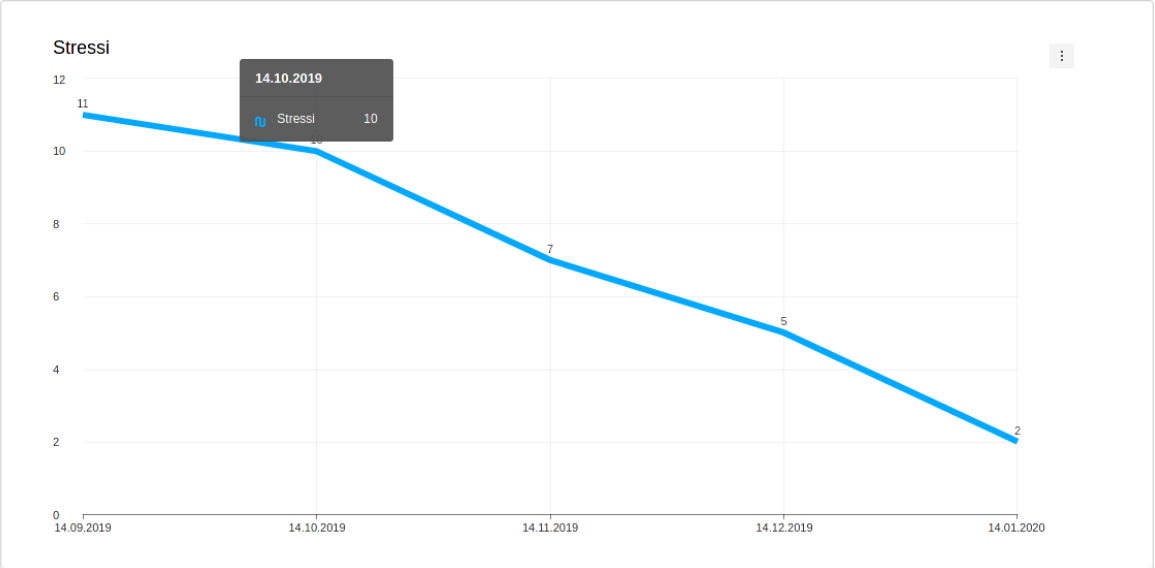
Maanantai 02.03.2020

Aloitin päiväni käymällä läpi edellisen viikon perjantaina kirjoittamani komponentin koodia. Rakennamme toisen opiskelijan kanssa Navisec Healthiin raportointiosiota, jossa voi luoda yksilö- ja massaraportteja erilaisista lomakkeista. Yksilöraporttiin sisältyy yhden asiakkaan tekemät täytöt yhdestä lomakkeesta. Lomake voi olla esimerkiksi fysioterapeutin asiakkaalleen teettämä kysely, jossa kartoitetaan asiakkaan fyysistä vointia. Lomake voi olla jaettu useaan osioon ja jokaisella kysymyksellä voi olla numeerinen arvo. Jokaisen osion vastauksista voidaan muodostaa osiolle tunnusluku (koodissa *indicator*), jonka laskukaavan lomakkeen luoja voi muodostaa haluamallaan tavalla. Jos terapeutin asiakas täyttää lomakkeen kuukauden välein puolen vuoden ajan, tunnuslukuja vertailemalla voidaan selvittää, onko asiakkaan tilanne muuttunut parempaan tai huonompaan suuntaan. Raportointiosiolla haluamme helpottaa tätä tulkintaa luomalla lomaketäytöistä kaavioita ja Excel-taulukoita.

Aloitimme yksilöraporteista, koska tunnistimme jo suunnitteluvaiheessa, että massaraportit tulevat olemaan monimutkaisempia toteuttaa, mutta tulevat sisältämään osittain samoja komponentteja. Saimme prototyypin kasaan odotettua nopeammin, ja tällä hetkellä lopullisesta tuotoksesta puuttuu enää suodattimet, joilla voi suodattaa raportilta tunnuslukuja tai täyttökertoja. Saimme tehtävän alussa Design-tiimiltä ulkoasusuunnitelman yksilöraporttinäkymästä, mutta suunnitelman suodattimet eivät mielestäni olleet käyttäjäystävällisiä ja ne rajoittivat näytettävän datan määrää. Lähetin Slack-kanavallemme ehdotuksen toisenlaisesta ulkoasusta, mutta tuoteomistajamme on lomalla, joten en ole saanut vahvistusta kummalla tavalla suodatus toteutetaan. Päätimme kuitenkin tiiminvetäjän kanssa, että lähden tekemään toteutusta oman näkemykseni mukaan. Suodattimien logiikka mahdollistaa kuitenkin kumman tahansa toteutuksen, joten vaihdon ei pitäisi tuottaa suurempia ongelmia.

Sain suodatuskomponentin melkein valmiiksi perjantaina, mutta tutkittuani koodia hetken huomasin, että voin yksinkertaistaa logiikkaa merkittävästi. Refaktoroin komponentin koodia, kunnes olin samassa tilanteessa kuin edellisen viikon lopussa. Refaktoroimalla sain poistettua kaksi funktiota, joten se tuntui jälkikäteenkin tarpeelliselta. Suodattimet ovat

raportointinäkymässä nappeja, jotka ovat päällä tai pois päältä. Jos mikään suodatin ei ole päällä, raportointinäkymä muodostaa jokaisesta osiosta (ts. tunnusluvusta) kaavion.



Kuva 2. Viivakaavion esimerkki.

Raporttien sisältämä data haetaan back-endistä, jossa se muotoillaan valmiiksi kaavioiden vaatimaan muotoon. GET-pyyntö lomakkeen tietoihin palauttaa listan täyttöpäivämääristä ja listan objekteja, jotka sisältävät jokaisen osion ja siihen liittyvät täyttöarvot. Tämän jälkeen nämä listat elävät kahden React -komponentin paikallisessa *statessa*. Raportoinnin päänäkymä hakee datan back-endistä ja passaa sen suodatinkomponentille propseina. Kun käyttäjä painaa jotain suodatusnappia, suodatuskomponentti siirtää valitun objektin toiseen listaan ja suodattaa objektin nykyisestä listasta pois.

```
// Filter indicator from 'indicators' to 'filteredIndicators'
filterIndicator (indicator: *) {
  const indicators = [ ...this.state.indicators ]
  const filteredIndicators = indicators
    .filter(item => item.id !== indicator.id)
  this.setState({
    indicators: filteredIndicators,
    filteredIndicators: [ ...this.state.filteredIndicators, indicator ]
  })
}
```

Refaktoroinnin päätteeksi toiminnallisuudet olivat siis tässä vaiheessa. Päivän varsinaisena tavoitteena oli luoda funktio suodatettujen objektien palauttamiselle, kun suodatin ote-

taan pois päältä. Tämän lisäksi tarvittiin vielä *callback* -funktio, joka päivittää datan päänäkymässä, kun suodatinnappeja aktivoidaan tai deaktivoidaan.

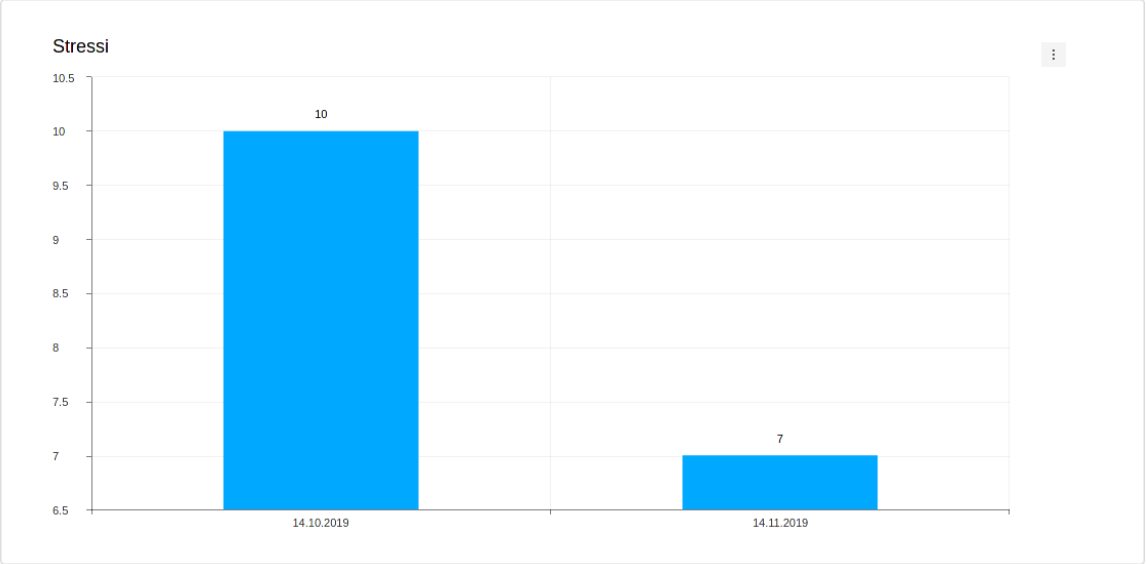
Suodatus toiseen suuntaan oli varsin helppo toteuttaa; lisäsin komponentille funktion, joka toimii samalla tavalla kuin yllä mainittu *filterIndicator* -funktio, mutta se siirtää objektin suodatettujen listalta takaisin päänäkymään lähtevään listaan. Siirrossa piti kuitenkin huomioida, että tunnusluvut ovat samassa järjestyksessä kuin alkutilanteessa. Luin pikaisesti Mozillan dokumentaatiota JavaScriptin *sort* -metodista ja loin nuolifunktion, joka järjestää tunnusluvut ID:n perusteella ennen callback -funktion kutsua.

Kaikki näytti jo toimivan kuten pitääkin, mutta callback-funktion kanssa oli vielä pieni ongelma. Ensimmäinen suodattimen aktivointi ei tehnyt mitään, mutta sitä seuraavat toimivat niin kuin pitääkin. Autoin harjoittelijaamme noin tunnin, joten ongelmanselvitys jäi seuraavalle päivälle.

Olin varsin tyytyväinen päivän tuloksiin, etenkin aamun refaktorointiin, jolla koodista tuli huomattavasti siistimpää. Lopun ongelma jäi mietityttämään, mutta olin melko varma, että saisin sen korjattua seuraavana päivänä.

Tiistai 03.03.2020

Tiesin, että päivästä tulisi rikkonainen ja lyhyt. Toivoin ehtiväni selvittää eiliseltä jääneen ongelman, joka paperilla vaikutti pieneltä, mutta jonka syy oli odotettua vaikeampi selvittää. Kävin läpi edellisenä päivänä kirjoittamaani koodia, ja arvelin ongelman olevan Reactin staten tallentumisessa. Päätin tulostaa logeja staten tilasta selaimen konsoliin. Kokeilin tehdä pieniä muutoksia staten hallintaan, mutta ongelma tuntui lähinnä pahenevan. Jouduin keskeyttämään debugauksen ja lähtemään viikkopalaveriin. Kenelläkään ei ollut juuri uutta esiteltävää palaverissa, joten sovimme vain, miten etenemme muun muassa massaraportoinnin kanssa. Massaraportointi on työkalu asiakasyrityksen myynnille, jolla voidaan helposti demonstroida terapian vaikutuksia lomaketäyttöihin perustuen. Jos esimerkiksi 30 fysioterapeutin asiakasta täyttävät jonkin fyysistä kuntoa kartoittavan lomakkeen viisi kertaa puolen vuoden aikana, massaraportilla muodostetaan kaaviot lomaketäyttöjen arvoista hoitojakson alussa ja lopussa.



Kuva 2. Pylväskaavion esimerkki.

Emme kuitenkaan ominaisuutta suunnitellessa koskaan määritelleet tarkemmin, miten lomaketäyttöjen alku- ja lopputilanteet haetaan tietokannasta. Tämän päivän tiimipalaverissa sovimme, että alkutilanne muodostetaan hakemalla jokaisen asiakkaan ensimmäisten lomaketäyttöjen tunnuslukuarvot ja lasketaan niistä keskiarvot kaavioita varten. Lopputilanne muodostetaan vastaavasti viimeisten lomaketäyttöjen tunnuslukuarvoista. Sovimme, että työkaverini lähtisi toteuttamaan Djangolla viewiä, joka palauttaa halutun datan front-endiä varten.

Itse jatkaisin yksilöraporttien parissa. Tiesin, että käyttämässämme kaaviokirjastossa useampi kaavio ottaa datan samassa muodossa kuin tähän asti käyttämämme viivakaavio. Ehdotin, että yksilöraportteihin lisättäisiin mahdollisuus valita kaavion tyyppi pudotusvalikosta raporttia luodessa. Ideaa pidettiin hyvänä, ja lupasin toteuttaa ominaisuuden seuraavan viikon palaveriin mennessä. Yksilöraporteista puuttui muun muassa vielä täyttökertojen suodatus ja raportin exportaaminen Excel-taulukoksi. Lupasin aloittaa myös näiden parissa, kunhan saisin tunnuslukusuodattimen toimimaan kunnolla.

Palaverin jälkeen lähdimme piakkoin lounastamaan. Lounaan jälkeen ehdin hetken etsiä ongelmaa tunnuslukusuodattimista ennen kuin harjoittelijamme pyysi apua Djangon testien kanssa. Korjasimme harjoittelijan kanssa yhden "rikki" menneen testin, ja opetin hänelle erilaisia tapoja ajaa testejä. Lähetin hänelle myös aikoinani tekemäni *VSCode-taskin*, jolla voi ajaa testejä näppäinkomennoilla. En ehtinyt auttaa muiden testien kanssa, sillä jouduin lähtemään hammaslääkäriin.

Torstai 05.03.2020

Jatkoin heti aamulla tunnuslukusuodattimen ongelman selvitystä. Olin nukkunut huonosti ja tunsin oloni turhautuneeksi, kun en ollut edennyt omissa töissäni muutamaan päivään. Olin edelleen vakuuttunut, että ongelma on Reactin setState-funktion käytössä, ja yritin etsiä netistä esimerkkejä sen oikeaoppisesta käytöstä. Pian huomasinkin, että luomassani *filter* -funktiossa, joka palauttaa suodatetun datan *child*-komponentilta *parent*-komponentille, state päivittyy selvästi väärin. Korjasin funktiossa olevan setState-funktion korvaamaan state oikein ja sain korjattua useamman päivän vaivanneen ongelman.

```
filter (filteredIndicators: Array<*>) {
    this.setState(prevState => {
      let fillData = Object.assign({}, prevState.fillData)
      fillData.series = filteredIndicators
      return { fillData, showCharts: false }
    })
  }
```

Poistelin kaikki lisäämäni logitukset ja testailin suodatinta varmistaakseni, että se toimii. Suodatin itsessään tuntui toimivan niin kuin pitääkin, mutta kaavioiden renderöitymisessä oli vielä ongelma. Kaaviot eivät päivittyneet reaaliaikaisesti, kun suodattimia lisätään tai poistetaan, koska jatkuva uudelleenrenderöinti on melko raskas toimenpide. Sen sijaan kaaviot päivittyvät vain, kun raportin muodostamisnappia painetaan. Jostain syystä kuitenkin, jos raportti oli jo muodostettu ja suodattimia laitettu päälle, ensimmäinen kaavio ei muodostusnappia painamalla päivittynyt ollenkaan. Päätin korjata ongelman piilottamalla kaaviot, jos suodattimia kytketään päälle tai pois päältä. Tällöin kaikki kaaviot luodaan aina alusta alkaen uudestaan, ja kaaviot renderöityivät oikein. Korjaus oli helppo toteuttaa, koska kaavioiden renderöityminen on yhdestä raporttinäkymän statessa sijaitsevasta boolean-arvosta kiinni. Mitä tahansa suodatinta painaessa tämä arvo asetetaan *falseksi*, ja kun raportin muodostusnappia painetaan, niin arvoksi tulee *true*. Esittelen toiminnallisuuden seuraavassa viikkopalaverissa, jossa voimme päättää onko tekemäni korjaus hyväksyttävä vai keksimmekö toisen tavan.

Iltapäiväni kului harjoittelijaamme auttaessa. Hän oli jo useamman viikon ajan toteuttanut Navisec Healthiin avainsanaominaisuutta, jossa eri komponenteille voidaan luoda avainsanoja, joilla komponentteja on helpompi suodataa erilaisissa listausnäkymissä. Ominai-

suus oli vasta-aloittaneelle harjoittelijalle varsin suuri, ja ongelmia olikin ollut paljon. Työkaverini oli pääasiassa vastuussa harjoittelijan opettamisesta, mutta hän työskenteli tänään etänä. Auttaminen oli omalta osaltani haastavaa, koska muutoksia oli edellisten viikkojen aikana luotu kymmeniin eri tiedostoihin, enkä itse ollut perillä mitä kaikkea oli muutettu. Keskityimme kuitenkin yhteen ongelmaan, joka liittyi avainsanojen suodatukseen. Katsoin, miten suodatus oli toisaalla tehty, ja kävin harjoittelijalle läpi, miten suodatuksessa käytettävä komponentti toimii ja miten se on kytketty *Reduxiin*, jossa suodatuksessa käytettävää tietoa säilytetään. Enempää en ehtinyt kuitenkaan tällä erää auttamaan, sillä olin sopinut menoa töiden jälkeen ja jouduin lähtemään.

Tunnuslukusuodattimen korjaaminen tuntui huojentavalta. Harmittelin, että en ehtinyt auttaa harjoittelijaamme enempää, mutta tiesin, että maanantaina voisimme katsoa ongelmia porukalla, jolloin auttaminenkin olisi helpompaa.

Perjantai 06.03.2020

Asetin päivän tavoitteekseni kaaviotyypin valintavalikon ja *Excel* -export -toiminnallisuuden luomisen. Tunnuslukusuodatuksen ongelmanratkonnan jälkeen kaipasin jotain helppoa tehtävää, joten aloitin kaaviotyypin valintavalikosta. Katsoin designista, miltä valikon haluttiin näyttävän, ja päätin käyttää samaa geneeristä *ChoiceField* -komponenttia, jota samassa näkymässä käytetään lomakkeen ja lomakkeen version valitsemiseen. Muutin Chart -komponentin toimintaa siten, että kaavion tyypin voi antaa komponentille propsina merkkijonon sijaan numerona, ja loin komponentin alkuun exportoitavan staattisen listan objekteista, jotka sisältävät tyypin id:n ja nimen.

```
// Ennen
export const CHART_TYPES = {
  line: 'line',
  column: 'column'
}


// Nyt
export const CHART_TYPES = [
  { value: 1, display_name: __('Line') },
  { value: 2, display_name: __('Column') },
]
```

ChoiceField pystyisi käyttämään tätä listaa suoraan vaihtoehtojen renderöintiin alasvetovalikossa, joten jäljelle ei jäänyt kuin tyypin lisääminen stateen ja sen passaaminen Chart-komponentille kaavioita luodessa. Mahdollistin alkuun kolme eri kaaviotyyppiä: viiva-, pylväs- ja palkkikaaviot. Näitä testaillessa päätin kuitenkin, että pylväs- ja palkkikaaviot ovat niin samanlaisia, että toinen ei olisi tarpeen. Palkkikaavion käyttö olisi myös vaatinut joitain dynaamisesti passattavia lisäasetuksia, jotta data olisi näyttänyt järkevältä, joten päätin luopua siitä kokonaan. Varmistin vielä, että tyypin vaihto onnistuu normaalisti kaikilla eri suodattimilla, ja latasin ominaisuuden Githubiin.

Lounaan jälkeen keskustelimme jälleen massaraporttien datan palautuksesta tiiminvetäjän ja massaraportteja tekevän kehittäjän kanssa. Olimme tähän mennessä palauttaneet datan muodossa, joka on suoraan yhteensopiva kaaviokirjaston kanssa. Tämä kuitenkin vaikeutti testausta, ja päätimme, että data palautettaisiin järjestelmällisemmässä muodossa back-endistä, ja muokataan raporteille sopivaksi front-endin puolella. Tämä ei vaatinut minulta välittömiä toimenpiteitä, mutta tiesin, että joutuisin todennäköisesti ensi viikolla luomaan yksilöraportteihin kyseisen muunnoksen.

```
// Nykyinen malli, suoraan yhteensopiva TUIChart-kirjaston kaavioiden kanssa
categories = ["01.02.2019", "01.03.2019", "01.04.2019"]
series = [
  { name: "Stressi", data: [9.6, 7.6, 5.5] },
  { name: "Ahdistus", data: [13.4, 8, 5.4] },
]

// Tuleva malli, helpompi testata, mutta täytyy kääntää kaavioille sopivaan muotoon
data = [
    {
        "date": "2019-02-01",
        "fills": [
            {"Stressi": 9.6},
            {"Stressi": 7.6},
            {"Stressi": 5.5}
        ]
    },
    {
        "date": "2020-03-01",
        "fills": [
            {"Ahdistus": 13.4},
            {"Ahdistus": 8},
            {"Ahdistus": 5.4}
        ]
    },
```

Iltapäivällä aloitin export -toiminnon kehityksen. Koko yksilöraportin data pitäisi voida exportoida joko Excel-taulukkona (.CSV, .XLS/.XLSX), tai PDF-tiedostona. Excel-taulukkoa luodessa ei tarvitse välittää suodattimista, joten sen voisi muodostaa suoraan back-endistä. Aloitin luomalla raportointinäkymään painikkeen, josta aukeaisi alasvetovalikko, josta export-tyypin voi valita. Sopivaa komponenttia etsiessäni sattumalta huomasin, että *Dropdown*-komponentille voi antaa propseina minkä tahansa nappikomponentin, joten se toimisi tarkoitukseeni erinomaisesti. Tämän takia olisi mielestäni tärkeää, että Dropdownin kaltaisten geneeristen komponenttien alussa olisi dokumentoitu kuvaus siitä, mitä komponentilla tehdään, ja esimerkiksi mitä propseja komponentille voi antaa. Laitoin itselleni muistutuksen, että lisäisin kyseisen dokumentaation komponentin kuvaukseen nykyisten toimeksiantojeni jälkeen.

Seuraavaksi lähdin etsimään netistä Python-pakettia, joka helpottaisi Excel-tiedostojen muodostamista Djangossa. Pythonista löytyy valmiiksi moduuli, jolla voi exportoida CSV-tiedostoja, mutta XLS-tiedostot tarvitsevat jonkin kolmannen osapuolen luoman paketin. Asensin *xlwt* -nimisen paketin, jolla lähdin luomaan viewiä, joka palauttaisi tyhjän, ladattavan Excel-taulukon. Xlwt:n dokumentointi oli todella suppeaa, mutta löysin pari blogikirjoitusta, joissa oli esimerkki, kuinka paketin avulla luodaan view joka palauttaa ladattavan XLS-tiedoston. Loin näkymälle urlin, ja kytkin sen front-endissä olevaan pudotusvalikkoon, joka lähettää pyynnön kyseiseen osoitteeseen. Pyyntö meni onnistuneesti läpi, mutta pyynnön sisältämä vastaus sisälsi vain pitkän merkkijonon eikä tiedostoa muodostunut. Selvittelin asiaa google-hauilla, mutta en löytänyt ratkaisua ongelmaan. Kysyin apua myös ainoalta paikalla olleelta kehittäjältä, mutta hänkään ei osannut sanoa mikä voisi olla vialla. Päätin kokeilla toista pakettia, ja asensin *openpyxl* -nimisen paketin. Openpyxl:n dokumentaatio oli huomattavasti laajempi, ja sen logiikka tyhjien tiedostojen muodostamiseen oli lähes identtinen xlwt:n kanssa. Lopputulos oli kuitenkin sama; tiedostoa ei muodostunut, kun viewiin lähettää GET-pyynnön. Kello oli jo varsin paljon, joten päätin jatkaa ongelmanratkaisua seuraavalla viikolla.

Olin tyytyväinen aamupäivän aikana kehittämääni kaaviotyypin valintavalikkoon, mutta viikon päättäminen täysin tuntemattomaan ongelmaan oli jokseenkin ärsyttävää.

Seurantaviikko 1 Analyysi

Ensimmäinen raportointiviikkoni oli varsin hajanainen. Toisaalta saavutin melkein kaikki tavoitteeni, mutta samalla tuntui, etten saanut paljoakaan aikaan. Varsinaista kehitystyötä hidasti sekä harjoittelijan auttaminen, viikkopalaveri että datarakenteiden suunnittelu toi-

sen kehittäjän kanssa. Tämän lisäksi yksi puolikas työpäivä ja yksi opinnäytetyön kirjoittamiseen käytetty päivä tekivät viikosta myös normaalia lyhyemmän.

Yksi selvästi esiin nouseva teema oli jälleen kerran dokumentaatio. Sekä Navisec Healthin että ulkopuolisten kirjastojen tai pakettien puutteellinen dokumentaatio hidastavat kehitystyötä huomattavasti. Tämän tunnistaminen on tehnyt itsestäni paremman dokumentoijan. Olen yrittänyt tartuttaa tapaa dokumentoida, ainakin uusia komponentteja luodessa, myös muille tiimiläisille. On myös tärkeää tunnistaa, milloin dokumentaatio on tarpeellista. Hyvä dokumentaatio on yksinkertaista ja tiivistä, ja se selittää vain tarvittavat asiat, eikä jokaista koodiriviä tarvitse tai kannata dokumentoida (Onatsu 2018, luku 4.1). Tässä riittää itselläni vielä varmasti opittavaa, mutta lähtökohtaisesti aion dokumentoida mieluummin vähän liikaa kuin liian vähän.

Vaikka Reactin staten hallinta on yksi Reactin kulmakiviä, ja olen tehnyt sitä kaikissa React-sovelluksissani jo useamman vuoden ajan, minulla riittää selvästikin opeteltavaa sen oikeaoppisesta hallinnasta. React suorittaa muutoksia state dataan asynkronisesti ja saattaa ryhmitellä useita muutoksia yhteen suorituskyvyn parantamiseksi, joka tarkoittaa, että setState-kutsun efekti ei välttämättä toimi odotetulla tavalla (Freeman 2019, osa 2, luku 11).

Vaikka ongelmia viikolla riitti, sain itsevarmuutta onnistumisista. Pienenkin ominaisuuden kehittäminen ns. "omasta päästä" tuo hyvää mieltä, ja helpottaa tulevien haasteiden kanssa painimista. Olenkin huomannut, että jään usein selvittämään jotain ongelmaa liian pitkäksi aikaa. Pitäisi osata tehdä välissä jokin pienempi taski, jottei pääsisi turhautumaan yhden ongelman pohdintaan.

3.2 Seurantaviikko 2

Maanantai 09.03.2020

Jatkoin maanantaina exportien parissa. Kuvittelin edelleen, että ongelma olisi helpohko ratkaista, ja voisin saada sekä Excel-, että PDF-exportit valmiiksi huomiseen palaveriin. Päätin yksinkertaistaa lähestymistapaani vielä vähän, ja koetin tehdä alkuun CSV-exportin. CSV:n muodostaminen Djangolla oli hyvin samantapainen prosessi kuin XLS-tiedoston muodostaminenkin; haetaan tietokannasta queryset, ja kirjoitetaan se tiedostoon. Tämän jälkeen tiedosto annetaan viewille liitteenä, jonka voi ladata tekemällä pyynnön sille määriteltyyn urliin. Tulokset olivat samoja kuin aiemmin; pyyntö meni läpi onnis-

tuneesti, mutta vastaus oli vain merkkijono, jossa oli CSV -tiedon sisältö. Jatkoin ongelman selvittelyä Googlen avulla, ja vihdoin selvisi, että kyseessä on JavaScript-ongelma: JavaScriptissä ei turvallisuussyistä sallita tiedostolatauksia.

Löysin ongelmalle useita ratkaisuita, mutta oikeastaan kaikki niistä olivat "hackeja", kuten näkymättömien linkkien luontia tai *iFramen* käyttöä. Kaikilla tuntui olevan hieman erilainen ratkaisu ongelmaan, mutta itse en saanut mitään ratkaisua toimimaan halutulla tavalla. Joillain tavoilla sain CSV-tiedoton ladattua, mutta jos muutin tyypin XLS:ksi, tiedosto oli korruptoitunut. Koska olin lähtenyt tekemään toteutusta sillä oletuksella, että kyseessä oli yksinkertainen homma, aloin tuntea oloni poikkeuksellisen turhautuneeksi. Koetin pyytää apua kahdelta muulta tiimiläiseltä, jotka olivat toimistolla, mutta hekään eivät osanneet äkkiseltään auttaa. Kyseessä ei kuitenkaan ollut äärimmäisen kiireellinen tehtävä, niin päätin, että voisi olla parempi seuraavana päivänä tehdä jotain muuta, ja lähestyä asiaa uudelleen myöhemmin vähemmän turhautuneena.

Vaikea päivä. Tavallaan tuntui, että opin uutta tiedostolatausten käsittelemisestä, mutta koska en saanut mitään kunnolla toimivaa ratkaisua aikaan, en selvästikään sisäistänyt ongelmaa kunnolla.

Tiistai 10.03.2020

Eilisten vastoinkäymisten takia en uskaltanut asettaa hirveästi tavoitteita päivälle. Päätin, että katson aamupäivällä hieman PDF-exporteja, ja viikkopalaverin jälkeen olisin viisaampi seuraavista tehtävistä.

Tiesin, että PDF-tiedosto pitäisi muodostaa front-endissä, sillä siinä pitäisi näkyä suodatettua dataa. Etsin React-kirjastoja PDF-tiedostojen muodostamiseen, ja kaksi selkeintä vaihtoehtoa oli *react-pdf* ja *react-to-pdf*. Ensin mainittu olikin jo asennettuna, mutta asensin silti myös toisen paketin, koska esimerkkien perusteella toteutus olisi sillä paljon yksinkertaisempi, mutta huonommin muokattavissa. Paketti oli ollut vasta vähän aikaa kehityksessä, eikä sillä ollut hirveästi viikoittaisia latauksia, joten en odottanut siltä paljoa.

PDF:n muodostaminen paketilla tapahtui luomalla referenssi *React.createRef()* -funktiolla div-elementtiin, jonka sisällä olisi tulostettava sisältö. Muodostin referenssin luodut kaaviot sisältävään elementtiin, ja sain ladattua PDF-tiedoston näkymästä. Tulosteessa kuitenkin näkyi vain pieni osa haluamastani näkymästä. Koetin muuttaa niitä vähiä muokkausase-

tuksia mitä paketilla oli, mutta tilanne ei juuri parantunut. Poistin paketin ja lähdin viikkopalaveriin.

Viikkopalaverissa kävimme läpi lähinnä raportteja koskevia asioita. Tiimikaverini esitteli luomansa Django-funktiot ja näkymät, joita käyttäisimme sekä massa-, että yksilöraporteissa. Voisin siis aloittaa tekemään perjantaina mainitsemaani datakonversiota yksilöraportteja varten. Sovimme myös lomaketäyttöön luotavasta uudesta osiosta, jonne luotaisiin näkymä, jossa yksittäiselle lomaketäytölle voi lisätä tiedon onko kyseinen täyttö jonkin hoitojakson alku- tai lopputilanne. Tämä helpottaa tulevaisuudessa massaraporttien muodostamista, kun esimerkiksi alkutilannetta ei tarvitse muodostaa hakemalla tietokannasta jokaisen asiakkaan kyseisen lomakkeen ensimmäistä oletettua täyttöä, vaan tilanne on suoraan merkittynä lomaketäytön modeliin, jolloin sitä voi suoraan käyttää suodattimena. Samassa näkymässä lomaketäytölle voisi lisätä myös metatietoja, joita myöhemmin tullaan käyttämään massaraporttien suodatuksessa. Muut sovitut tehtävät olivat pienempiä ulkoasumuutoksia lomakeraporttinäkymiin.

Iltapäivällä kävimme tiimikaverini kanssa läpi datakonversion toteutusta. Päätimme, että tekisimme front-endiin kaksi *utility*-funktiota, joilla Djangosta saatu data ensin jaettaisiin kahteen settiin, joita olisi helppo käyttää täyttökertojen ja tunnuslukujen suodatukseen. Suodatuksen jälkeen raporttia muodostaessa toinen utility-funtio yhdistäisi suodatetun datan ja kääntäisi sen suoraan taulukoille sopivaan muotoon.

Aloin tuntemaan oloni kipeäksi, joten otin työkoneen mukaani ja lähdin kotiin vähän suunniteltua aikaisemmin. En tänäänkään edistynyt merkittävästi minkään homman parissa, mutta seuraavien päivien tehtävät olivat nyt selvillä.

Keskiviikko 11.03.2020

Tunsin oloni aamulla edelleen kipeäksi, mutta ajattelin ainakin aloittaa datakonversion työstämisen. Rajapinnasta pyydettävässä datassa tunnuslukudata käyttää Pythonin *DefaultDictia*, joka ei aiheuta *KeyErroria*, jos siitä yrittää hakea objektia, jota ei ole olemassa, vaan luo automaattisesti uuden "oletusobjektin". Halusin käyttää jotain vastaavaa ensimmäisessä datakonversiofunktiossani, joten aloin selvittelemään onko JavaScriptissä mitään vastaavaa toiminnallisuutta. Pian sain selvitettyä, että JavaScriptin Proxy-objekti toimii suunnilleen vastaavalla tavalla. Loin esimerkistä uuden Proxy-objektin:

```
class DefaultDict {
  constructor (defaultInit) {
  return new Proxy({}, {
  get: (target, name) => name in target ?
    target[name] :
      (target[name] = typeof defaultInit === 'function' ?
        new defaultInit().valueOf() :
        defaultInit)
    })
  }
}
```

Proxya käyttämällä sain ensimmäisen datakonversiofunktion muuttamaan serveriltä saadun datan suodattimille sopivaan muotoon varsin kivuttomasti. Oloni kuitenkin heikkeni sen verran, että jätin toisen funktion rakentamisen myöhemmälle.

Seurantaviikko 2 Analyysi

Lyhyeksi jääneeltä viikolta ei jäänyt paljoakaan käteen. Exportit osoittautuivat odotettua hankalammiksi toteuttaa, enkä etenkään Excel-exportien kohdalla sisäistänyt kunnolla ongelmaa, tai miten sitä voisi ratkaista. Ominaisuus ei kuitenkaan ollut erityisen kiireinen, joten tein niin kuin olen aikaisemmin todennut parhaaksi; vaihdoin tehtävää ja palaisin asiaan myöhemmin, toivottavasti uusien ideoiden ja näkökulmien kanssa.

Ensimmäisen puolivuotiseni aikana olen huomannut, kuinka vaikeaa ohjelmointitöiden vaativuutta tai ajankäyttöä on määritellä. Moni tehtävä saattaa tuntua hyvin yksinkertaiselta ja nopealta toteuttaa, mutta pieneen bugikorjaukseenkin saattaa kulua tunteja pelkästään sen toimivuuden testaamisen takia. Joskus pieni bugikorjaus voikin muuttua jonkin ominaisuuden uudelleenkirjoittamiseksi, jolloin voidaan helposti puhua tunnin sijaan päivästä. Henkilökohtainen arviointi todennäköisesti helpottuu kokemuksen karttuessa, mutta muuttujia on aina niin paljon, että tarkkoja arvioita tuskin pystyy koskaan antamaan. Olen kuitenkin huomannut, että esimerkiksi viikkopalavereissa jonkin tehtävän arviointi on helpottunut, jos useampi osallistuja ottaa siihen kantaa. Parempia arvioita voidaan saavuttaa perustamalla niitä kaikkien kokemuksiin. Useamman henkilön osallistuminen arviointiin antaa ohjelmoijalle paremman kuvan siitä, minkälaisia haasteita voi odottaa (Gera 2017, luku 4).

Uutta ohjelmistoa kehittäessä arvioita ei yleensä ole tarvinnutkaan antaa erityisen tarkasti. Uusien ominaisuuksien kehityksessä voidaan puhua viikoista tai kuukausista, ja suurin

osa bugeista ja korjauksista aikataulutetaan seuraavaan testi- tai tuotantojulkaisuun. Kriittisiin bugeihin pitää tietenkin varautua kiireellisellä aikataululla, ja niitä korjatessa on erityisen tärkeää hyödyntää muiden tiimiläisten osaamista ja kokemusta. Epätarkka aikataulutaminen ei myöskään aina tarkoita, että tehtäviin menee suunniteltua enemmän aikaa. Joskus tehtävään saattaa löytyä valmiit komponentit tai hyvä kirjasto, joka voikin muuttaa viikon työn päivän työksi.

3.3 Seurantaviikko 3

Tiistai 17.03.2020

Saimme viikonlopun aikana tiedotteen, että koronaviruspandemian takia työskentelisimme toistaiseksi etänä. Tiimillämme on aina ollut täydet valmiudet työskennellä etänä, joten tilanteessa ei varsinaisesti ole mitään ihmeellistä. Olen itsekin pitänyt etäpäiviä silloin tällöin, mutta olen jo opiskeluaikoina oppinut, että kotona on usein vaikeampi keskittyä opiskeluun tai työntekoon. Nyt tilanteeseen oli kuitenkin sopeuduttava, sillä se tulisi todennäköisesti kestämään kuukausia.

Pidimme viikkopalaverin Google Meetin kautta. Saimme kuulla, että nykyisen pandemiatilanteen takia Navisec Healthin ja muidenkin talossa kehitettävien etäterapiapalveluiden kysyntä oli viime viikkojen aikana kasvanut huomattavasti. Tämä tulisi todennäköisesti muuttamaan työskentelyjärjestelyitämme väliaikaisesti, mutta kuulisimme asiasta tarkemmin seuraavana päivänä. Pidimme palaverin lyhyenä, ja sovimme, että kaikki jatkavat nykyisiä tehtäviään tässä vaiheessa.

Tavoitteeni päivälle oli saada toinen datakonversiofunktio valmiiksi, ja sen avulla saada molemmat raporttisuodattimet toimimaan. Funktion pitäisi yhdistää kaksi suodatettua dataobjektia yhteen, ja välittää se kaaviokomponentille raporttia muodostaessa. Vaikka kyseessä oli melko tavallinen datarakenteiden käsittelyä vaativa toimenpide, proxyn käyttäminen aiheutti hieman päänvaivaa. Päätin lähteä aluksi tekemään mitä tahansa toimivaa ratkaisua, ja keskittyä suorituskykyyn ja koodin siisteyteen myöhemmin. Koodin testaaminen oli varsin hidasta, koska raporttinäkymän avaaminen vaatii pari toimenpidettä käyttöliittymässä. Docker -kontissa pyörivä kehitysympäristö kävi myös aika hitaalla, joskin uudelleen käynnistäminen auttoi asiaa jonkin verran.

Muutamassa tunnissa sain aikaan funktion, joka muutti datan haluamaani muotoon, ja siirryin tekemään varsinaista suodatuksen koodia. Tunnuslukusuodattimen koodi toimi

suoraan uuden datasetin kanssa, joten lähdin kopioimaan samaa logiikkaa täyttökerta-suodattimille. Hetken näytti jo, että logiikka toimi heti ensimmäisellä yrityksellä, mutta suodattimien yhtäaikainen käyttö aiheutti outoja ongelmia, jotka aiheuttivat datan nollautumisen. Tunsin kuitenkin päässeeni suurin piirtein tavoitteeseeni ja päätin jatkaa korjauksia seuraavana päivänä.

Keskiviikko 18.03.2020

En halunnut aloittaa päivääni heti ensimmäisenä bugikorjauksella, joten päätin käyttää kuukausittaiset opiskelutuntini ostamaani Django-kurssiin. Saamme käyttää kuukaudessa kaksi tuntia omaehtoiseen opiskeluun, joka auttaa meitä suoriutumaan paremmin työssämme. Lisäksi saamme ostaa erilaisia kursseja tai kirjoja yrityksen piikkiin. Kyseessä on mielestäni todella tärkeä etu, ja olenkin ehdottanut kuukausittaisten opiskelutuntien lisäämistä, mutta toistaiseksi asiaa ei olla käsitelty sen enempää. Hankkimallani Django-kurssilla luodaan suhteellisen yksinkertainen kiinteistöapplikaatio Djangolla ja *PostgreSQ*L:llä. Aloittelijoille tarkoitettu kurssi ei ole juuri tarjonnut itselleni uutta tietoa, mutta haluan suorittaa kurssin loppuun ennen seuraavaan siirtymistä.

Puoliltapäivin pidimme tiimin tilannekatsauksen. Koronaviruspandemia oli lisännyt etäterapian kysyntää paljon, ja resursseja ja prioriteetteja muutettiin yrityksen sisällä vauhdikkaasti. Meidän tiimimme kohdalla tämä tarkoitti sitä, että toisessa tiimissä kehitetty Viivi Health-sovellus siirtyisi meidän tiimimme kehitettäväksi. Viivi Health on sessiopohjainen videoneuvottelupalvelu, joka on integroitu useaan talossa kehitettävään ohjelmistoon. Oma työnkuvani ei juuri muuttunut tilanteen takia, mutta lomakeraportteja kanssani tehnyt tiimikaverini siirtyisi lähiaikoina kehittämään Viiviä, joten jatkaisin raporttien kehittämistä yksin. Joutuisin todennäköisesti reagoimaan myös bugiraportteihin enemmän nyt kun tiimin resursseja oli järjestelty uudelleen.

Palasin tutkimaan suodattimien ongelmaa, mutta en ehtinyt kauaakaan tutkimaan asiaa, kun tiimikaverini ilmoitti, että hän oli joutunut tekemään muutoksia back-endista lähetettävään querysetiin. Muutos oli tavallaan pieni, mutta Pythonin *DefaultDictistä* luopuminen tarkoitti, että datakonversiofunktiot menisivät uusiksi. Vaikka muutos oli perusteltu ja vastaavia tilanteita tulee alalla vastaan usein, tuntui todella turhauttavalta palata melkein takaisin lähtöruutuun tehtävän kanssa, jonka kanssa oli paininut jo lähes viikon ja viimein melkein saanut valmiiksi. Jouduin vielä odottelemaan, että muutokset tulisivat Gitiin, joten kävin läpi tekemästäni koodista osia, joita voisin mahdollisesti vielä hyödyntää. Lopulta päätin säästää kaikki tekemäni muutokset varmuuden vuoksi. Iltapäivällä sain haettua

muutokset Gitlabista, ja mergesin konfliktit hyväksymällä kaikki olemassa olevat ja tulevat muutokset.

Torstai 19.03.2020

Tarkastin back-endiin tehdyn muutoksen ja aloin työstämään uusia datakonversiofunktioita. Dataobjektit eivät olleet enää "anonyymejä", vaan nimi ja arvo saatiin nyt eriteltynä. Tämä tarkoitti sitä, että JavaScriptin Proxya ei tarvitsisi enää käyttää. Tehtävää helpotti myös se, että olemassa oleva logiikkaa datan yhdistämiseen voisi hyödyntää, vaikka koodi menisikin uusiksi. Tällä kertaa en halunnut testata muutoksia kehitysympäristössä, vaan päätin hyödyntää yhtä monista internetin koodisandboxeista. CodePen (https://codepen.io/) tuki modernia JavaScript-syntaksia (ei tarvitse käyttää esimerkiksi puolipisteitä koodirivien lopussa), joten päädyin käyttämään sitä. Aloitin luomalla kaksi apufunktiota datan yhdistämiseen:

```
function mergeArrays (arrays) {
  return [].concat.apply([], arrays)
}

function groupBy (data, key, valuesKey) {
  return data.reduce((obj, item) => {
    const group = item[key]
    const value = valuesKey ? item[valuesKey] : item
    obj[group] = obj[group] || []
    obj[group].push(value)
    return obj
  }, {})
}
```

Täyttökertasuodattimille datan pystyi jälleen antamaan lähes suoraan. Apufunktioita ja ES6-funktioita hyödyntämällä data oli suhteellisen helppoa muuntaa tunnuslukusuodattimien haluamaan muotoon:

```
const fillData = responseData.map(submission => submission.fills)
const mergedFills = mergeArrays(fillData)
const grouped = groupBy(mergedFills, 'name', 'value')
const indicators = Object.entries( grouped ).map( ([ name, data ]) => ({ name, data }) )
```

Testailin koodia CodePenissä, ja kaikki näytti toimivan hyvin. Yllä mainittua logiikkaa käyttämällä data oli myös helppo muuttaa kaavioille sopivaksi. Siirsin koodit varsinaiseen ap-

plikaatioon, ja jouduin vielä tekemään pieniä muutoksia, joita en tajunnut ottaa huomioon CodePenin puolella. Pienen säädön jälkeen ilokseni huomasin, että molemmat suodattimet toimivat niin kuin pitääkin. Tein vielä nopean katsauksen voisiko jotain logiikkaa lyhentää tai uudelleen käyttää, mutta en äkkiseltään löytänyt mitään, joten latasin muutokseni gitiin.

Vihdoinkin pääsin pisteeseen, jossa halusin olla jo melkein viikko sitten. En toki voinut ennustaa back-endiin tulevaa muutosta tai aikaisempaa sairastelua, mutta tuntui hyvältä päästä etenemään työtehtävissä.

Seurantaviikko 3 Analyysi

Tällä viikolla saatiin taas yksi oppitunti siitä, miten hyvä suunnittelu voi säästää aikaa. Back-endiin tehty muutos oli pieni, mutta omalla kohdallani se tarkoitti yhden työpäivän edestä "hukkaan heitettyä" työtä. Kaikkea ei voi toki ennustaa etukäteen, mutta hyvin suunniteltu API helpottaa front-end-kehittäjän elämää merkittävästi. Tämän viikon odottamaton muutos kuitenkin helpotti back-endin testaamista ja selkeytti front-endin datakonversiofunktioita, joten uudestaan tehdystä työstä huolimatta se tuntui hyödylliseltä.

Samalla kun viikon mittaan pyörittelin datakonversiofunktioita ja datasuodattimia, huomasin taas ajautuneeni versionhallinnan kannalta epäoptimaaliseen asemaan. Olin tehnyt paljon muutoksia useaan tiedostoon, mutta en ollut tehnyt Git commiteja missään välissä. Huomasin pariinkin kertaan kaipaavani jotain pientä osaa koodia, jonka olin tehnyt aiemmin, mutta koska en ollut laittanut sitä versionhallintaan, vaan tehnyt muutokset vain paikallisesti, en päässyt siihen enää käsiksi. Git Towerin Git-oppaassa (Learn Version Control with Git) listataan hyvin Git commitien parhaat käytännöt:

- Commitoi olennaiset muutokset. Commitin pitäisi toimia kääreenä siihen liittyville muutoksille. Jos esimerkiksi korjataan kaksi erillistä bugia, pitäisi syntyä kaksi erillistä niille kohdistettua commitia.
- Commitoi usein. Commitoimalla usein jaat koodia säännöllisemmin muille tiimin jäsenille. Näin kaikkien on helpompi integroida omia muutoksiaan välttäen merge-konfliktit.
- Älä commitoi puoliksi valmista työtä. Commitien tulee sisältää vain valmista työtä. Se ei kuitenkaan tarkoita sitä, että commitin pitäisi sisältää vaikkapa kokonainen

uusi ominaisuus. Ominaisuuden voi todennäköisesti pilkkoa pienemmiksi loogisiksi palikoiksi, jotka voi commitoida kun ne valmistuvat.

- Testaa ennen commitointia. Pientä muutosta tehdessä testaaminen on helppo unohtaa. Pienelläkin muutoksella voi kuitenkin olla sivuvaikutuksia, joten testaamiseen kannattaa käyttää hetki aikaa ennen commitin tekemistä.
- Kirjoita hyvä commit-viesti. Commit-viestin pitäisi selittää lyhyesti ja ytimekkäästi, mitä muutoksia commitilla tehdään. Selkeiden commitien viestit voidaan pitää yksirivisinä, mutta tarvittaessa commit-viestiin voi lisätä pidemmänkin viestin, jolla selitetään, miksi muutos on tehty.

Itselleni tasapainottelu toisen ja kolmannen kohdan välillä on ollut ajoittain vaikeaa. Jos teen muutoksia useaan eri paikkaan, saattavat muutokset kasautua yhdeksi commitiksi, vaikka niitä voisi jakaa pienempiinkin osiin. Aina tämä ei ole haitaksi, mutta välillä ongelmia tulee vastaan ja kehitystyö venyy ja commitit muuttuvat liian isoiksi.

3.4 Seurantaviikko 4

Maanantai 23.03.2020

Raporttien työstäminen jatkui. Sovimme edellisen viikon palaverissa, että raportin kaavioiden tulisi käyttää asiakkaan määrittelemää teemaväriä, ja fontti tulisi olla sama kuin muualla Navisec Healthissa. Fonttikokoa haluttiin myös kasvattaa selkeyden vuoksi. Navisec Healthissa hyödynnetään Djangolle luotua *django-tenants*-pakettia, joka tekee uusien asiakkaiden luomisesta helppoa. Sen sijaan, että jokaiselle uudelle asiakkaalle pystytettäisiin kokonaan uusi projekti, django-tenantsin avulla uusille asiakkaille voidaan luoda jaettuun tietokantaan oma tietokantataulu. Asiakkaat (*tenant*) voidaan tunnistaa uniikista isäntänimestä (esim. demo_tenant.navisec.fi) ja sekä tämä, että muut asiakkaita koskevat tiedot sijaitsevat *public* -nimisessä tietokantataulussa. Navisec-tiimillä on käytössään selaimessa toimiva käyttöliittymä asiakkaiden hallintaa varten. Asiakkaat puolestaan voivat kustomoida ominaisuuksiaan omista asetuksistaan. Näihin asetuksiin kuuluu muun muassa teema, jota voi kuitenkin tällä hetkellä hallita vain värin ja logon osalta.

Suurin osa asiakkaan määrittelemistä asetuksista sijaitsevat Reactin Redux-storessa, ja ovat siten helposti haettavissa propseina mille tahansa komponentille. *TUIChart*-kaavioiden ulkoasua pystyy muokkaamaan antamalla kaaviolle options -objektin, jossa on määritelty teema. Teeman voi määritellä yleisellä tasolla koko kaaviolle tai vain yksittäisille

osille. Jostain syystä en kuitenkaan saanut määriteltyä yleistä teemaa, vaan jouduin asettamaan fontit ja värit jokaiselle tarvittavalle osalle erikseen.

Teemoja muokatessa tajusin, että raporttinäkymää voisi selkeyttää, jos suodattimet eivät olisi näkyvillä ennen kuin tarjolla on suodatettavaa dataa. Asetin suodattimet renderöivään metodiin ehtolausekkeen, joka estää renderöinnin, jos statessa oleva dataobjekti on tyhjä. Vastaavia ehtolausekkeita käytetään paljon komponenttien render -metodeissa pitämään näkymät siisteinä.

Keskiviikko 25.03.2020

Lomaketäyttönäkymän asetusosio jäi työkaveriltani kesken, kun hän siirtyi muihin tehtäviin. Tavoitteenani oli saada näkymä koko lailla valmiiksi tämän päivän aikana. Lomaketäytön asetuksissa lomaketäytölle voi asettaa tilan, joka kertoo, onko lomaketäyttö asiakkaan ensimmäinen täyttö (alkutilanne) tai viimeinen täyttö (lopputilanne). Terapeutit voivat hyödyntää tilatietoa omassa työssään, mutta me voimme myös hyödyntää sitä massaraportteja luotaessa. Tällä hetkellä voimme hakea alkutilanteet raporttia varten vain hakemalla tietokannasta jokaisen asiakkaan ensimmäisen lomaketäytön, joka ei välttämättä ole oikea alkutilanne. Erikseen merkitty tilanne tekee hausta varman ja parantaa sen suorituskykyä. Lomaketäytön tilaksi voi määrittää myös väliarvion, mutta tämä on toistaiseksi vain terapeutteja varten.

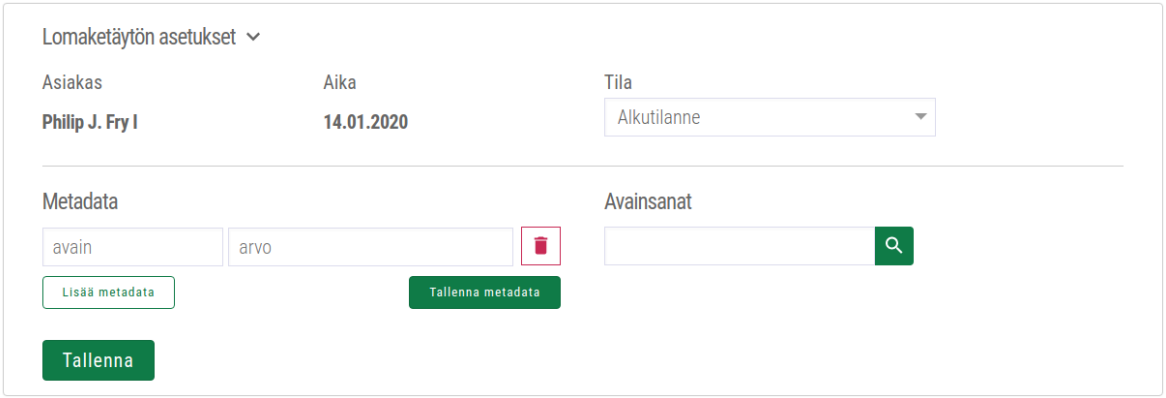
Tilavalikko oli jo tehtynä, mutta huomasin sattumalta, että se toimi vain pääkäyttäjällä. Työntekijäkäyttäjä näki vain tyhjän listan, ja ensimmäinen ajatukseni oli, että käyttöoikeuksissa oli jokin ongelma. Löysinkin tilanteen aiheuttavan ehtolausekkeen äkkiä, ja koodieditorini Git-lisäosa osasi kertoa, että työkaverini oli lisännyt tämän äskettäin. Varmistin häneltä, ettei kyseessä ollut toivottu toiminnallisuus ja korjasin ehtolausekkeen. Gitin versiohistorian näkeminen suoraan koodieditorissa on ollut suuri apu viimeisen vuoden aikana.

Tilavalikon lisäksi asetusosioon piti lisätä mahdollisuus lisätä lomaketäytölle metadataa ja avainsanoja. Avainsanaominaisuuden kehitys oli vielä kesken tiimimme harjoittelijalla, joten se pitäisi lisätä myöhemmin. Metadatalle sen sijaan olin tehnyt jo aikaisemmin talvella geneerisen komponentin, joka olisi helppo lisätä melkein mille tahansa ominaisuudelle. Back-endin puolella metadata hyödyntää Djangon *ContentTypea*, joka pitää kirjaa applikaation modeleista ja mahdollistaa geneeriset suhteet niiden välille. Metadatan lisääminen

lomaketäytöille back-endissa tarkoitti siis sitä, että lomaketäyttöjen urleihin lisättiin vain uusi url, jonka näkymälle passataan lomaketäytön model:

```
path('lomakkeet/<int:pk>/taytot/<int:taytto_id>/metadata/', MetadataView.as_view(model=Taytto)),
```

Näin Django osaa yhdistää API-pyynnöissä urlissa olevan id:n näkymän mukana menevään modeliin. Front-endin puolella metadatan lisääminen ei ollut juuri sen vaikeampaa. Metadata-komponentti odottaa saavansa propseilla urlin, johon se lisää metadataosuuden loppuun. Lisäsin nämä paikalleen ja metadata toimi niin kuin pitäisikin. Itse metadata syötetään tekstikentillä avain-arvo-pareina, joita voi lisätä periaatteessa loputtoman määrän. Muutin vielä metadata-komponentin kokoa, mutta muuten asetusosio alkoi näyttää valmiilta.



Kuva 3. Lomaketäytön asetusosio

Asetusosio alkoi viemään varsin paljon tilaa näkymästä, työntäen täytön varsinaisia tietoja pois näkymästä. Vain osa käyttäjistä tulisi käyttämään asetusosiota, joten sen olisi mielestäni hyvä olla oletuksena kutistettu. Muistin aikaisemmin nähneeni, että asennettujen JavaScript-pakettien listalla oli *react-slidedown* -niminen komponentti. Vilkaisin react-slidedownin dokumentaatiota, ja totesin, että se kävisi tarkoitukseeni mainiosti. Slidedownin teko oli todella helppoa; liu'utettava osio piti vain kääriä paketin tarjoamaan SlideDown-komponenttiin, ja asettaa stateen tieto, onko komponentti auki vai kiinni. Lisäsin osion otsikon perään vielä ikonin, josta käyttäjän oli helppo tulkita, että osion voi avata ja sulkea.

Päivän työtehtävät sujuivat tällä kertaa mutkitta, oli mukava huomata, että talvella tekemäni metadata-komponentti toimi tässäkin tapauksessa ilman ongelmia.

Perjantai 27.03.2020

Palasin lomakeraporttien pariin. Huomasin näkymässä heti ongelman. Testidatana käyttämäni lomaketäytöt ovat kuukausia sitten tehtyjä, joten niiden ei pitäisi näkyä heti näkymään tultaessa, koska näkymään haetaan oletuksena vain viimeisen 30 päivän tiedot. Olin keskiviikkona asettanut yhden lomaketäytön tilaksi alkutilanteen, ja se oli päivittänyt täytön päivämäärän. Haluamme kuitenkin säilyttää alkuperäisen täyttöpäivämäärän, ettei raporttien datan järjestys mene sekaisin.

Ongelma oli helppo ratkaista, koska lomaketäytöistä tallentuu tietokantaan kaksi päivämäärää, joista toinen on alkuperäinen luontipäivämäärä, ja toinen on viimeksi tehdyn muokkauksen päivämäärä. Tiedot haetaan raporteille tällä hetkellä edellisen muokkauksen päivämäärän mukaan. Luontipäivämäärän käyttäminen ei kuitenkaan ollut ongelma, joten kävin vaihtamassa tietokantakyselyn käyttämään luontipäivämäärää. Testailin vielä, että lomaketäytöt pysyivät oikeassa järjestyksessä niiden tietoja muokatessa, ja pushasin korjauksen Gitiin.

Nyt kun lomaketäyttöjä ei löytynyt viimeisen 30 päivän ajalta, tajusin, että näkymä saattaa näyttää uudelle käyttäjälle todennäköisesti oudolta. Lomake ja sen versio on valittuna, mutta mitään dataa ei ole näkyvillä. Päätin lisätä kenttien alle ilmoituksen, joka ilmoittaa, että valituilla päivämäärillä ei löydy lomaketäyttöjä. Tämän pitäisi ohjata käyttäjä muuttamaan päivämääräsuodattimia. Heti kun dataa löytyy, ilmoitus poistuu näkyvistä, ja suodattimet tulevat sen tilalle. Tehtävään löytyi valmis *Notification*-komponentti, jolla voi lisätä näkymään viestin. Viestille voi määrittää taustavärin, joka käyttää Bootstrap-kirjastostakin tuttuja danger (punainen), warning (keltainen), success (vihreä) ja info (sininen) värityksiä. Päätin käyttää keltaista väriä, joka ilmaisee, että kyseessä ei varsinaisesti ole virhe, mutta käyttäjältä odotetaan toimia.

Iltapäivällä osallistuin Zoomin välityksellä pidettyyn kuukausittaisen perjantaipalaverin, jossa tiedotettiin koronaviruspandemian vaikutuksista yrityksen taloustilanteeseen. Palaveri venyi yli tunnin mittaiseksi, ja keskittyminen työskentelyyn oli ollut koko päivän normaalia vaikeampaa, joten päätin lopettaa viikkoni hieman suunniteltua aikaisemmin.

Seurantaviikko 4 Analyysi

Ensimmäinen kokonainen etätyöviikkoni eteni suhteellisen mukavasti. Kotoa käsin työskennellessä keskittymiskyky on jatkuvasti koetuksella, enkä olettanutkaan, että olisin saman tien orientoitunut tekemään etätöitä samalla tavalla kuin toimistolla. Vaikka laitteistoni soveltuu täysin etätyön tekemiseen (useampi näyttö, vastamelukuulokkeet), on erilaisia häiriötekijöitä silti enemmän läsnä. Tilanne oli kuitenkin uusi monelle muullekin yrityksen työntekijälle, joten tilannetta helpotettiin jakamalla hyviä vinkkejä etätyön tekemiseen Slackin etätyökanavalla.

Sain jaettua viikkoni työt hyvin eri päiville. Valitsin kaavioiden teemamuutokset maanantaille, koska omasin jo hyvän käsityksen siitä, miten toteuttaisin ne. Tuntui hyvältä idealta aloittaa ensimmäinen etätyöviikko tehtävällä, jonka kanssa ei pitäisi tulla suurempia ongelmia. Aikaa kuitenkin kului työpisteen säätämiseen ja vähän muuhunkin ylimääräiseen, joten tehtävän suorittamiseen kului odotettua enemmän aikaa.

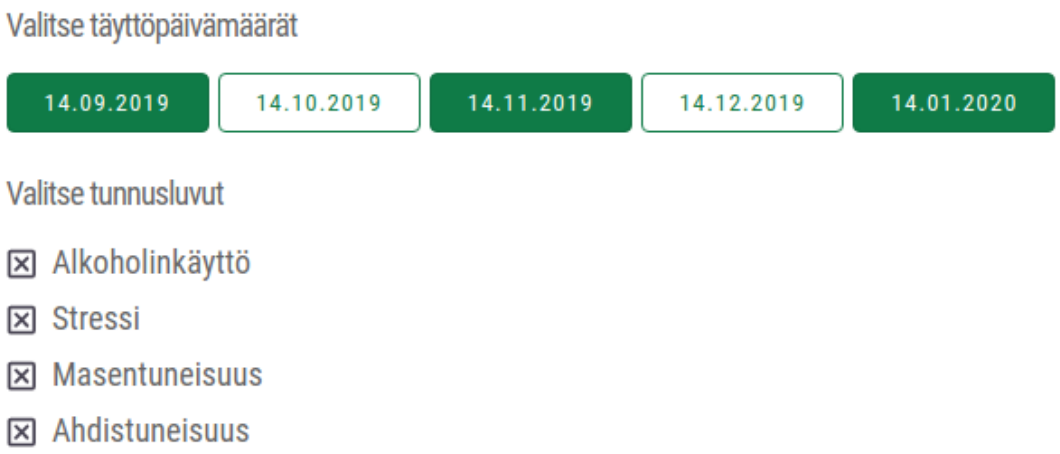
Keskiviikkona sain tehtyä lomaketäytön asetusnäkymän kuntoon, ja olin hyvin iloinen huomatessani, että aiemmin talvella tekemäni geneerinen Metadata-komponentti toimi juuri niin kuin pitääkin. Komponentti oli geneerinen sekä Djangon, että Reactin osalta. Geneeristen komponenttien konsepti oli toki itselleni ennestään tuttu juuri Reactin takia, koska React kannustaa luomaan uudelleenkäytettäviä käyttöliittymäkomponentteja, jotka esittävät muuttuvaa dataa (Hunt, 2013). Djangossa en ollut kuitenkaan aikaisemmin tehnyt geneerisiä komponentteja. Ryhmänjohtajamme kehotuksesta tutustuin Djangon ContentTypeihin, joiden yksi käyttötapaus on luoda geneerisiä suhteita modelien välille (Freitas, 2016). Käyttämällä ContentTypeja, olin onnistunut luomaan komponentin, jonka pystyi lisäämään sekä back- että front-endissä toiseen komponenttiin kirjoittamalla yhteensä vain kolme riviä koodia.

Perjantaille en ollut suunnitellut mitään tiettyä tehtävää, joten keskityin korjaamaan bugeja ja parantamaan raporttinäkymän käytettävyyttä. Olen nauttinut varsin vapaista aikatauluista raporttinäkymän kehityksessä. Toki nyt kun kehittäisin ominaisuutta yksin, aikataulut venyisivät entisestään. Sain kuitenkin edelleen kehittää ominaisuutta rauhassa, jolloin bugit on helpompaa huomata ajoissa ja käytettävyyteen ehtii panostaa paremmin.

3.5 Seurantaviikko 5

Tiistai 31.03.2020

En ollut tyytyväinen suodattimien ulkoasuun, ja ajattelin kokeilla vaihtoehtoista ratkaisua, jota voisin esitellä viikkopalaverissa. Tällä hetkellä käytettävistä painikkeista oli mielestäni hankalaa erottaa, mitkä painikkeiden osoittamista elementeistä olivat mukana suodatuksessa ja mitkä eivät. Vilkaisin komponenteista löytyvää *Checkbox*-komponenttia, joka voisi ajaa saman asian. Se olisi luontainen tapa valita asioita listalta ja varmasti selkeä loppukäyttäjälle. Sen käyttäminen vähentäisi myös koodirivejä huomattavasti. Pieni ongelma oli kuitenkin se, että checkboxit pitäisi listata allekkain, että ne näyttäisivät hyvältä. Ajattelin kuitenkin, että tässä voisi hyödyntää lomaketäytön asetuksissa käyttämääni SlideDown-komponenttia, jolla listan saisi piiloon, jos ei halua käyttää suodattimia. Korvasin toisen suodatinrivin checkboxeilla, mutta en vielä lisännyt toiminnallisuuksia niihin, kun en vielä tiennyt haluttaisiinko niitä käyttää.



Kuva 4. Suodatinvaihtoehdot. Yllä tähän asti käytetty painikemalli, alla uusi ehdotukseni.

Viikkopalaverissa esittelin vaihtoehtoiset suodattimet, mutta ne saivat melko vaisun vastaanoton. Sovimme, että ottaisin näyttökuvan ja kysyisin mielipidettä design-tiimiltä. Esittelin samalla viimeistelemäni lomaketäytön asetusosion, joka puolestaan sai hyvää palautetta. Palaverin jälkeen lähetin näyttökuvan suodattimista design-tiimille, ja vastausta odotellessani palasin korjaamaan pari asiaa lomaketäytön asetuksista, jotka huomasin palaverin aikana.

Käytän kehitysympäristööni luomiani tenanteja yleensä pääkäyttäjänä, koska pääkäyttäjällä on aina eniten ominaisuuksia käytössään. Kaikki uudet ominaisuudet pitää kuitenkin muistaa testata myös henkilökunta- ja asiakaskäyttäjillä. Palaverin aikana muistin, että metadataan oli asetettu sitä luodessa rajoitteet, että sitä pystyy käyttämään vain pääkäyttäjät. Nyt kun se liitettiin lomaketäyttöihin, pitäisi myös henkilökuntakäyttäjien päästä sitä

käyttämään. Metadata kyllä näkyi näkymässä henkilökuntakäyttäjälle, mutta pyyntöjen tekeminen oli estetty back-endistä. *Permission*-luokkien käyttäminen Djangossa on yksinkertaista:

```
class CanEditMetadata(permissions.BasePermission):
    def has_permission(self, request, view):
        # Aikaisemmin: if not request.user.is_superuser:
        if not request.user.is_staff:
            return False
        else:
            return True
```

Samalla kun korjasin käyttöoikeudet back-endiin, piilotin myös asetusnäkymän front-endissä asiakaskäyttäjiltä. Iltapäivällä tein vielä pientä refaktorointia asetusnäkymään siirtämällä inline-tyylejä css-luokkiin.

Keskiviikko 01.04.2020

Halusin tämän päivän aikana saada valmiiksi raporttinäkymän käyttäjähallinnan ja mainosnäkymän prototyypin. Toistaiseksi kehitysympäristössäni rapottinäkymä on ollut käytössä kaikilla käyttäjillä. Testipalvelimelle tai tuotantopalvelimelle päivitettäessä näkymän pitäisi kuitenkin olla näkyvillä vain työntekijöille, joilla on käytössään pro-palvelumalli. Palvelumalleja on käytössä tällä hetkellä kaksi, standard ja pro. Aloitin hahmottelemalla päässäni, montako erilaista käyttötapausta raporttinäkymään liittyisi. Käyttötapauksia oli kolme:

- Asiakaskäyttäjät (client, esim. terapeutin potilas) eivät saisi nähdä raportointinäkymää ollenkaan.
- Henkilökuntakäyttäjät (staff, esim. terapeutti), joiden palvelumalli on standard, saisivat nähdä näkymän, jossa esitellään raportointiominaisuus ja kehotetaan tutustumaan pro-tilaukseen (mainosnäkymä).
- Henkilökuntakäyttäjät ja pääkäyttäjät (admin), joiden palvelumalli on pro, näkisivät normaalin raportointinäkymän.

Lomakenäkymä on jaettu välilehtiin, joista yksi välilehti on nimeltään raportit. Aloitin lisäämällä välilehden otsikon perään kuvakkeen, joka ilmoittaa välilehden sisältävän Pro-käyttäjille tarkoitettua sisältöä. Tarkoitukseen oli olemassa valmis *Badge* -komponentti, mutta sen lisääminen otsikon perään ei ollut niin yksinkertaista kuin voisi kuvitella. Väli-

lehdet, kuten moni muukin sisältö, luodaan näkymään dynaamisesti JavaScriptin *map*-funktiolla. Jos välilehden komponenttia muuttaa suoraan, muuttuvat kaikki applikaation välilehdet. Annoin siis välilehtinäkymälle *badge-propin*, jolla välilehdelle voi halutessaan määritellä kuvakkeen. Jos badgea ei määritellä, välilehti renderöityy normaalisti.

Seuraavaksi lähdin toteuttamaan välilehden piilottamista asiakaskäyttäjiltä. Välilehdet annetaan välilehtikomponentille listana JavaScript-objekteja, joissa määritellään välilehden otsikko ja näkymä, joka välilehden alle aukeaa. Pystyin helposti hakemaan Reactin *contextista* tiedon, onko nykyinen käyttäjä asiakas- vai henkilökuntakäyttäjä. Tarkistin ehtolausekkeella käyttäjän mallin ja lisäsin välilehtilistaan raporttinäkymän, jos käyttäjä on henkilökuntakäyttäjä.

```
// Välilehtinäkymälle annettavat välilehdet
let items = [
  {
    label:   'Forms',
    content: <ListFormsView />,
  },
  {
    label:   'Submissions',
    content: <ListSubmissionsView />,
  }
]

if (context.loggedInAsStaff)
  items.push(
    {
      label:   'Reports',
      content: <FormReportView />
    }
  )
```

Lopuksi tein ulkoasusuunnitelman perusteella mainosnäkymän standard -käyttäjille. Näkymässä oli vain otsikko, lyhyt esittelyteksti ja esimerkkikuva kaaviosta, joten tehtävä oli suoraviivainen. Jäljelle jäi näkymän laittaminen näkyville henkilökuntakäyttäjille, joilla ei ollut pro-tilausta. Lisäsin aikaisempaan välilehtinäkymään samanlaisen ehtolausekkeen, jossa tarkistin Redux-storessa sijaitsevan käyttäjän tilaustason ja lisäsin näkymän listaan, jos ehto täyttyi.

Pääsin hyvin päiväni tavoitteisiin ja suoriuduin kaikista tehtävistä pitkästä aikaa kokonaan ilman mitään ulkopuolista tiedonhakua. Mainosnäkymä tosin oli vasta prototyyppi ja siihen tulisi varmasti muutoksia vielä myöhemmin.

Torstai 02.04.2020

Työkaverini oli edellisiltana laittanut viestiä, että hänellä ei näkynyt raporttikaavioissa ollenkaan kuvaajia. Arvelin ongelman liittyvän viikko sitten tekemiini teemamuutoksiin. Vaihdoin tenantini teemavärin pois päältä, jolloin tenant käyttää Navisecin vaaleansinistä oletusväriä. Loin raportin ja huomasin, että kaavioista tosiaan puuttuu kuvaajat. Olin unohtanut määrittää vaihtoehtoisen värin, jos Redux storesta ei löydy tenantille määriteltyä teemaväriä. Tämä oli helppo korjata. Laitoin oletusvärin aluksi suoraan merkkijonona, mutta äkkiä tajusin, että kannattaa käyttää staattista muuttujaa. Näin pidän huolen, että jos oletusväriä joskus muutetaan, niin se muuttuu automaattisesti myös kaavioille:

```
series: {
  // Asetetaan väriksi Redux storesta tenantille asetettu väri TAI oletusväri.
  colors: [ this.props.color || DEFAULT_PRIMARY_COLOR ]
}
```

Aamupalaa syödessäni luin blogikirjoitusta PDF-exportien luomisesta Reactilla, ja löysin siitä mielenkiintoisen vaihtoehdon, jota päätin kokeilla. Tarvitsisin kaksi JavaSciprt-kirjastoa:

- Html2canvas-kirjastolla voi ottaa näyttökuvia DOM-elementeistä. Näyttökuvat voi renderöidä suoraan toisen elementin sisään tai muuntaa kuvatiedostoiksi.
- jsPdf-kirjastolla voi luoda ladattavan PDF-tiedoston.

Voisin siis toisella kirjastolla ottaa raporttinäkymästä näyttökuvan ja antaa sen toiselle kirjastolle, joka luo siitä ladattavan PDF-tiedoston. Kävin läpi molempien kirjastojen dokumentaatiota ja sain varsin nopeasti luotua oikean kuvan sisältävän PDF-tiedoston. Kuva oli kuitenkin venynyt, eikä kaikki kaaviot mahtuneet PDF-arkille. Etsin netistä tietoa, miten kuvan voisi skaalata oikean kokoiseksi. Samalla löysin ohjeet, miten sain sivulta ylimääräiset valkoiset alueet pois ja sain metodini tekemään lähes oikeanlaisen PDF:n:

```
printPDF () {
  // Lasketaan dokumentin koko dynaamisesti sen perusteella, montako kaaviota näkymässä renderöidään.
  const count = this.state.chartData.series.length
  const height = 107.5 * count
  // Valitaan alue josta otetaan näyttökuva.
  const domElement = document.getElementById('report')
  // Luodaan näyttökuva html2canvas-kirjastolla.
  // Annetaan samalla asetukset jotka poistavat ylimääräiset valkoiset alueet.
  html2canvas(domElement, { scrollX: 0, scrollY: -window.scrollY })
  .then((canvas) => {
      const imgData = canvas.toDataURL('image/png')
      // Luodaan uusi PDF-dokumentti ja määritetään sen koko.
      const pdf = new jsPdf('p', 'mm', [ height, 210 ])
      // Skaalataan kuva oikein.
      const pdfWidth = pdf.internal.pageSize.getWidth()
      const pdfHeight = (imgProps.height * pdfWidth) / imgProps.width
      // Annetaan raportista otettu kuva jsPdf-kirjastolle.
      pdf.addImage(imgData, 'PNG', 0, 0, pdfWidth, pdfHeight)
      // Tallennetaan tiedosto ja käynnistetään sen lataus.
      pdf.save('form-report.pdf')
  })
}
```

Ladattu tiedosto näytti muuten oikealta, mutta kaavioihin ilmestyi rumia harmaita reunuksia. Nämä johtuivat todennäköisesti kaaviokirjastosta, joten niistä ei välttämättä pääsisi eroon. Päätin kuitenkin jättää toiminnallisuuden koodiin, ja esitellä sitä seuraavassa viikkopalaverissa.

Olin pettynyt lopputulokseen, kun luulin jo keksineeni ratkaisun PDF-exportien luomiseen. Kyseessä oli todellakin oletettua suurempi ongelma. Voisin kuitenkin saada mielipiteitä ja kehitysehdotuksia seuraavassa viikkopalaverissa.

Seurantaviikko 5 Analyysi

Sain tälläkin viikolla melko paljon aikaan, mutta lopputulokset jättivät toivomisen varaa. Tiistaina yritin olla oma-aloitteinen ja loin vaihtoehtoisen ulkoasun suodattimille. En onneksi kuitenkaan luonut sille toiminnallisuuksia kokonaan, sillä siitä ei innostuttu, ja myöhemmin siitä luovuttiin kokonaan. Niistä luopuminen oli myös hyvin perusteltu, sillä jos lomakkeella olisi esimerkiksi kymmeniä täyttökertoja, checkboxeista muodostuva lista olisi liian pitkä, vaikka sen saisikin piilotettua. En kuitenkaan antanut tämän haitata liikoja, vaan siirryin testaamaan sovellustamme. Emme testaa front-endiä ollenkaan Reactin omilla testeillä tai minkään kirjaston avulla, vaan testaaminen tapahtuu manuaalisesti klikkailemalla. En ole koskaan kokeillut Reactin testejä, mutta olen lukenut ristiriitaista informaatiota siitä, mitä Reactilla olisi hyödyllistä testata siihen kuluvaan aikaan nähden. JavaScriptin testaamisen asiantuntija Kent C. Dodds (2019) summaa asian lyhyesti: "Kirjoita testejä. Ei liian montaa. Lähinnä integraatioon." Olen halunnut itsekin tutustua Reactin testaamiseen,

ja uskon, että meidän tiimimmekin voisi hyötyä testaamisesta. Ymmärrän kuitenkin myös, että pienessä tiimissä ilman vanhempia React-osaajia testaamiseen todennäköisesti kuluisi liikaa aikaa siitä saatavaan hyötyyn nähden.

Tiistaina tehtyjen käyttäjätasotestien kautta päätin keskiviikon aikana hoitaa lomakeraporttien käyttäjätasot ja mainosnäkymän prototyypin kuntoon. Tehtävät olivat varsin suoraviivaisia, ja mainosnäkymäkään ei jäänyt varsinaisesti prototyypiksi, sillä siitä haluttiin myöhemmin vaihtaa vain mainoksena toimiva kuva. Tuntui hyvältä tehdä koko päivä töitä ilman ulkopuolista tiedonhakua. Keskittyminenkin oli helpompaa, kun pysty pysymään kehitysympäristön sisällä monta tuntia putkeen.

Torstaina sen sijaan jouduin tekemään ulkopuolista tiedonhakua sitäkin enemmän, kun palasin PDF-exportin pariin. Löysin sattumalta blogitekstin, jossa lueteltiin kolme tapaa luoda PDF-dokumentti React-sivusta. Ensimmäinen tapa herätti mielenkiintoni: Jos PDF:n ei tarvitse olla yhteensopiva valittavan tai haettavan tekstin kanssa, yksinkertaisin tapa luoda PDF on ottaa näyttökuva sivusta, ja asettaa se PDF-tiedostoon (Boér, 2019). Blogissa mainittiin myös tähän tarvittavat kirjastot, ja esimerkkikoodi yksinkertaisen PDF:n tekemisestä. Jouduin vielä tukeutumaan varsin paljon molempien kirjastojen dokumentaatioon, mutta lopulta sain PDF-tiedoston aikaiseksi. Se ei kuitenkaan näyttänyt hyvältä, joten joutuisin palaamaan sen pariin myöhemmin uudelleen. Seuraavalla viikolla alkaisin toteuttaa kokonaan uutta toimintoa, joka oli kiireellisempi kuin PDF-export.

3.6 Seurantaviikko 6

Tiistai 07.04.2020

Aloitin työstämään toimintoa, jolla minkä tahansa luodun kaavion voi lähettää asiakkaalle viestillä. Lopullisessa versiossa lähetyksen tulisi toimia niin, että kaaviosta otetaan kuva png-muodossa, ja lisätään se automaattisesti näkymään. Ensimmäinen versio päätettiin kuitenkin toteuttaa niin, että asiakas lataa itse kuvan, ja lisää sen viestin liitteeksi. Kaaviosta sai kuitenkin jo ladattua hyvän kuvan TUIChartin tarjoamalla export-toiminnolla. En osannut määritellä kauanko ensimmäisen version valmistumiseen menisi, mutta en odottanut sen valmistuvan vielä tämän päivän aikana.

Tarkistin näkymälle tehdyt ulkoasusuunnitelmat, joissa näkymä oli modaalin sisällä. Lisäsin jokaisen generoidun kaavion alle napin, josta modaalin voi aukaista. Käytämme *react-awesome-modal* -kirjastoa, mutta olemme rakentaneet oman modaalikomponenttimme

sen päälle. Modaalin lisääminen näkymään oli helppoa; importataan komponentti, lisätään se render-metodiin ja lisätään stateen tieto, onko modaali auki vai kiinni. Lisäsin nämä ensin raporttinäkymän luokkakomponenttiin, mutta luokan koko ylitti jo 400 riviä, joten päätin lisätä modaalin omaan luokkaansa. Raporttinäkymä tarvitsi vain tiedon siitä, onko modaali auki vai ei, muuten modaalin logiikka tuntui järkevältä eristää omaan luokkaansa.

Ensimmäinen asia, jota lähdin lisäämään modaaliin oli pudotusvalikko, jossa listataan keskustelut, joissa viestin lähettäjä ja valittu asiakas ovat osallisina. Ajattelin joutuvani lisäämään back-endiin endpointin joka palauttaisi haluamani keskustelut, mutta sieltä löytyi jo tarpeeseeni sopiva url. Se vaatisi pieniä muokkauksia, mutta riitti tässä vaiheessa hyvin. Endpointia testatessa kuitenkin huomasin, että viestien lähetys ei toiminut. Vikailmoitus viittasi siihen, että migraatioissa oli ongelmaa. Olin hiljattain vaihtanut branchia versionhallinnassa, ja epäilin, että siellä tehty Djangon migraatio oli luonut tietokantaan kolumnin, joka puuttui omalta branchiltani. Löysin virheen aiheuttavan kolumnin ja vaihdoin branchin samaan, jossa olin aiemmin migraatiot tehnyt. Etsin migraation, jossa kenttä oli lisätty ja ajoin migraatiot sitä edeltävään vaiheeseen. Vaihdoin branchin takaisin omaani ja viestien lähetys toimi jälleen normaalisti.

Lisäsin modaaliin ChoiceField-komponentin, jolle annoin vaihtoehdoiksi back-endista saadut keskustelut. Keskusteluissa oli melko paljon tietoja, jota en tarvinnut, joten käytin JavaScriptin map -funktiota niiden karsimiseen:

```
async getThreads () {
  // Haetaan keskustelut palvelimelta
  const response = await api.get('/threads/')
  if (response.ok) {
    // Poistetaan keskusteluista muut tiedot paitsi id ja keskustelun aihe
    const threadChoices = response.data.map(thread => ({
      value: thread.id,
      display_name: thread.subject,
    }))
    this.setState({ threadChoices, selectedThread: threadChoices[0].value })
  }
}
```

Lopuksi varmistin, että keskustelun vaihtaminen toimii oikein. Lisäsin pudotusvalikon viereen vielä napin, jolla voisi myöhemmin aloittaa uuden keskustelun, jos listasta ei löydy sopivaa keskustelua. Etenin päivän aikana mukavasti ja olin tyytyväinen migraatioongelmien nopeaan ratkomiseen.

Keskiviikko 08.04.2020

Jatkoin viestimodaalin tekoa. Tunsin itseni jo aamulla todella väsyneeksi ja töiden aloittaminen oli vaikeaa. Keskittymiskyky oli olematon, mutta yritin jatkaa siitä, mihin jäin edellisenä päivänä. Tällaisia päiviä on onneksi harvoin, mutta väistämättä näitäkin tulee joskus vastaan.

Halusin ensimmäisenä saada modaaliin tekstikentän viestille. Tähän ei löytynyt mitään varsinaista valmista komponenttia, mutta käytämme kenttien (ja lomakkeiden) rakentamiseen paljon *fieldsets* -kirjastoa. Voimme luoda kenttiä lähettämällä back-endille options-pyynnön, jonka datasta kentät muodostuvat:

```
// Options-pyyntö back-endiin
async setFieldset (): Promise<void> {
  const fieldset: * = await getFieldset(this.url, 'POST', false, this.onError)
  this.setState({ fieldset })
}

// Back-endin palauttama data, tässä tapauksessa viesti-objektin 'body', eli varsinainen viesti
actions > POST > body:
{
  type: "string",
  required: true,
  read_only: false,
  label: "Viesti",
  max_length: 4095,
  help_text: null
}
```

Back-endin palauttama data voidaan renderöidä näkymään tekemällämme *Field*-komponentilla, jolle voidaan määritellä minkälaista tekstikenttää haluamme käyttää:

```
const fieldname = 'body'
const Field = this.state.fieldset.fields[fieldname]
<Field
  value={ this.getValue(fieldname) }
  errors={ this.getErrors(fieldname) }
  onChange={ this.onChangeBody }
  component={ MultilineTextField }
  props={{ maxLines: 3 }} />
```

Tämä ei ole ainoa tapa käyttää fieldsettejä. Niille on rakennettu kokonaan oma näkymäkin, mutta tässä tapauksessa tarvitsisin maksimissaan kaksi kenttää, joten uskoin tämän olevan hyvä toteutustapa. Jouduin hieman palauttamaan mieleeni, miten fieldsetit toimivat, mutta viestikentän lisääminen näkymään oli varsin mutkatonta.

Sen sijaan options-pyynnössä käytettävän urlin päivittämisessä dynaamisesti oli yllättäviä ongelmia. Halusin päivittää pyynnössä lähtevään urliin keskustelun ID:n aina kun keskustelua vaihdetaan pudotusvalikosta. Tämä oli kyllä helposti saatavilla statesta, mutta jostain syystä se ei toiminut. Lisäsin luokalle *componentDidUpdate* -elinkaarimetodin, jossa voisin tarkistaa, milloin pudotusvalikosta vaihdetaan keskustelua. Onnistuin kuitenkin aiheuttamaan päättymättömän loopin, joka kaatoi koko selaimeni. Mietin kauan, mitä olin tehnyt väärin, kunnes huomasin, että kutsun modaalin tallennusmetodia (joka ei vielä tehnyt mitään muuta kuin kirjasi konsoliin tekstiä) väärin. Typerä virhe, mutta päivän keskittymistason huomioon ottaen en ollut yllättynyt.

Samalla kun korjasin ID:n vaihdon, tajusin, että voisin siirtää myös options -pyynnön tekevän kutsun componentDidUpdate-metodin sisään, ettei sitä kutsuttaisi turhaan, ennen kuin modaali avataan. Sen jälkeen päätin, että on aika lopetella tältä päivältä.

Torstai 09.04.2020

Edellisen päivän väsymyksen merkit olivat edelleen vahvasti läsnä ja tuleva pääsiäisloma tulisi selvästi tarpeeseen. Edessä oli viestimodaalin hankalin osuus: kuvaliitteen lisääminen. Liitetiedostojen käsittelyyn on tavallisesti käytetty kahta eri lähestymistapaa:

- Tiedosto tallennetaan Djangon *FileField* tai *ImageField* -kentällä, joka tallentaa tiedoston johonkin määriteltyyn kansioon, ja säilöö siitä referenssin, jolla tiedosto voidaan halutessaan hakea. Tämä on helpoin ja nopein tapa käsitellä tiedostoja ja on kätevä esimerkiksi asiakkaan (tenant) logon lisäämiseksi, koska logoja voi olla vain yksi. Jos tiedostoja alkaa kuitenkin kertyä paljon, kuluttavat ne järjestelmän fyysistä muistia merkittävästi.
- Tiedosto tallennetaan pilvipalvelimelle, jossa tallennetun tiedoston ID ja URL tallennetaan tietokantaan. Tällöin tiedostojen määrällä ei juuri ole väliä, koska pilvipalvelimen resurssit eivät kuluta järjestelmän fyysisiä resursseja. Tiedostot kuitenkin ainakin Navisec Healthin tapauksessa tallentuvat omaan tietokantatauluunsa, joten ne pitää yhdistää oikeisiin modeleihin.

Kävin läpi olemassa olevia komponentteja, joiden avulla tiedostoja muissa osioissa hallitaan. *AttachmentField* -kenttää käytettiin viestinäkymässä ja siinä olisi ollut yllä mainitun pilvipalvelin-esimerkin logiikka valmiina. Se ei kuitenkaan ulkoasunsa puolesta ollut optimaalisin vaihtoehto, etenkään kun halusin saada lähetettävästä kuvasta esikatselukuvakkeen viestikentän viereen. Toinen komponentti, *FileSelectField* taas oli ulkoasunsa puolesta täydellinen, mutta sen logiikka oli tehty Djangon FileFieldille. Joutuisin siis joko luomaan kokonaan uuden komponentin tai yhdistämään olemassa olevien komponenttien logiikkaa.

Kokeilin molempia komponentteja, mutta turhauduin melko nopeasti, kun tuntui, ettei mikään toiminut niin kuin haluaisin. Päätin lähestyä asiaa uudestaan pääsiäisloman jälkeen, toivottavasti uusilla ideoilla ja paremmin levänneenä.

Seurantaviikko 6 Analyysi

Haastava viikko. Tavoitteenani oli saada viestimodaali valmiiksi jo tämän viikon aikana, mutta aliarvioin oman osaamiseni, jaksamiseni ja vaaditun työn määrän. Edistystä kuitenkin syntyi ja tuleva pääsiäisloma auttaisi taas jaksamaan paremmin.

Pääsin tiistaina hyvään alkuun luomalla modaalin viestin lähetystä varten. Olin myös iloinen siitä, että sain ratkaistua migraatioista johtuneen virheen omin avuin. Migraatioongelmat voivat olla vaikeita tunnistaa, mutta jos virheviesti kertoo, että tietokannasta puuttuu jokin kenttä, on kyseessä todennäköisesti migraatioista johtunut virhe. Tämän jälkeen täytyy vain tunnistaa ongelman aiheuttanut migraatio, ja ajaa migraatiot taaksepäin sitä edeltävään vaiheeseen. Sain lisättyä myös keskusteluvalinnan, joka ei aiheuttanut ongelmia, koska olin viime aikoina tehnyt vastaavia pudotusvalikoita enemmänkin. Tässä toki auttavat hyvät uudelleenkäytettävät komponentit, kuten käyttämäni ChoiceField.

Keskiviikkona jouduin muistuttamaan itseäni siitä, miten fieldsetit toimivat. Näille onneksi löytyi dokumentaatiosta hyvät ohjeet, joilla sain viestikentän toteutettua ongelmitta. Myöhemmin kuitenkin onnistuin kaatamaan koko selaimeni Reactin elinkaarimetodien kanssa. componentDidUpdate -elinkaarimetodin sisällä oleva koodi tekee jotain aina, kun komponentissa oleva data päivittyy. Sillä on siis erityisen helppoa saada aikaan "infinite loop", joka suorittaa koodia toistuvasti niin kauan, että käyttäjä keskeyttää sen vaikkapa sulkemalla selaimen.

Torstaina en juurikaan edistynyt tehtävissäni. Väsyneenä helpotkin tehtävät saattavat tuntua ylitsepääsemättömiltä, jos ei nopeasti saa tehtävästä kiinni. Tiesin kyllä mitä halusin saavuttaa, ja löysin kaksi komponenttivaihtoehtoa mitä voisin käyttää, mutta kumpikaan niistä ei toiminut niin kuin odotin. Väsyneenä komponenttien toimintalogiikan selvittämien kävi ylivoimaiseksi.

Ei sellainen viikko, mitä halusin. Olin kuitenkin optimistinen, että loman jälkeen voisin virkein mielin saada viestimodaalin valmiiksi.

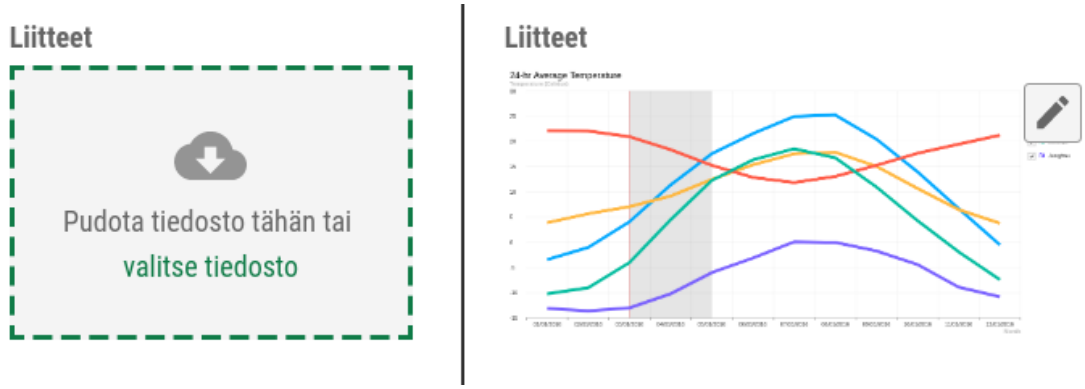
3.7 Seurantaviikko 7

Tiistai 14.04.2020

Ehdin pääsiäisloman aikana pohtia, miten raporttiviestin liitteet voisi toteuttaa. Ideanani oli käyttää FileSelectField-komponenttia, jolle lisäisin metodin, joka tallentaa kuvan liitteenä. Tallentamisen jälkeen komponentti palauttaisi modaalille tallennetun liitetiedoston id:n, jonka voisi lisätä viestin POST-pyyntöön viestiä lähetettäessä, jolloin liitteen pitäisi tallentua viestiin.

Aluksi minun piti käydä läpi, miten FileSelectField toimii, ja miten data sen sisällä liikkuu. Komponenttia käytettiin Navisec Healthin harjoiteosiossa, jossa voidaan luoda erilaisia moniosaisia harjoitteita potilaille. Jokaiselle harjoitteen osalle oli komponentin avulla mahdollista lisätä opastava kuva tai video. Päätin käyttää komponenttia, koska se luo esikatseluversion lisätystä tiedostosta, ja tällaista oli ulkoasusuunnitelmassa toivottu. Avasin harjoitteen, jolle lisäsin uuden kuvan ja samalla seurasin selaimeni *React Devtools*-lisäosalla, mitä komponentissa tapahtui. Toistin tämän muutaman kerran ja uskoin päässeeni hyvin jäljille. Kopioin komponentin logiikan modaaliini ja kokeilin mitä tapahtuisi, jos yritän lisätä kuvan. Komponentti näytti siltä miltä pitäisi, mutta tiedoston valinta ei tuntunut tekevän mitään. Tutkin asiaa jälleen React Devtoolsilla, tällä kertaa kahdella eri välilehdellä. Modaalin komponentti ei näyttänyt muuttavan arvoaan kuvan valinnan yhteydessä.

Seuraavat pari tuntia käytin logaamalla erilaisia muuttujia ja tekstejä konsoliin yrittäen selvittää, missä vaiheessa modaalin komponentin datankulku katkesi. Aloin jo vaipua epätoivoon, mutta huomasin viimeinkin metodin, joka ei asettanut kuvan tietoja stateen oikein. Korjasin metodin ja testatessani kuvan lisäämistä esikatselukuva tuli näkyviin, joten tulkitsin, että komponentti toimi niin kuin pitääkin.



Kuva 5. Kuva FileSelectField-komponentista ennen kuvan lisäämistä, ja kuvan lisäämisen jälkeen.

Tajusin samalla, ettei minun tarvitsisi tehdä muutoksia FileSelectField-komponenttiin suoraan vaan voisin tehdä kuvan lähettämisen modaalin sisällä. Kuvaa ei myöskään tarvinnut tallentaa ennen esikatselukuvan luontia vaan sen voisi tallentaa viestin lähetyksen yhteydessä. Lisäsin viestin lähetysmetodiin logiikan, jossa kuva tallennetaan ensin liitteenä, jonka jälkeen liitteen id lähetetään luotavalle viestille. Tämän jälkeen Django osaa yhdistää viestin ja kuvan seuraavasti:

```
async sendMessage () {
  // Jos kuva on lisätty, lähetetään se liitteiden endpointiin
  if (this.state.file) {
    const fileData = new FormData()
    fileData.append('file', this.state.file)
    const response = await api.post('/attachments/', fileData)
    if (response.ok)
      this.setState({ id: response.data.file.id })
  }
  // Annetaan viestille itse viesti, ja liitetiedoston id
  const data = { body: this.state.body, attachment_ids: [ this.state.id ]}
  const response = api.post(this.url, data)
}
```

Testasin kuvallisen viestin lähetystä, ja status -koodit näyttivät, että kaikki olisi mennyt oikein. Jännittyneenä menin viestiosioon ja avasin viestiketjun, jonne olin viestin lähettänyt. Viesti ja siihen liitetty kuva olivat menneet perille oikein. Tein vielä pieniä säätöjä ja refaktorointeja, mutta olin huojentunut, että sain kuvan lähettämisen toimimaan.

Keskiviikko 15.04.2020

Vaikein osuus viestimodaalin ensimmäisestä versiosta oli valmiina, mutta pienempiä muutoksia ja lisäyksiä oli vielä paljon tehtävänä. Toivoin saavani ainakin suurimman osan niistä tehtyä tämän päivän aikana. Ensimmäisenä piti päättää, mitä eri näkymiä käyttäjä näkee riippuen siitä, haluaako hän lähettää viestin olemassa olevaan keskusteluun vai luoda uuden keskustelun. Uuden keskustelun luomisen logiikka puuttui myös vielä kokonaan.

Ihan ensimmäiseksi lisäsin keskusteluvalikkoon tyhjän vaihtoehdon. Tähän asti valikkoon on haettu kaikki keskustelut ja valituksi keskusteluksi on asetettu listan ensimmäinen keskustelu. Tämä toimii vain sillä oletuksella, että keskusteluja on olemassa. Tyhjä vaihtoehto varmistaa, että virhettä ei tapahdu, vaikka yhtään olemassa olevaa keskustelua ei löytyisi. Tämä mahdollisti myös kahden eri näkymän valinnan; jos valitsee listasta keskustelun, aukeaa näkymä, jossa viestiin voi lisätä vain itse viestin ja kuvan. Jos taas painaa 'Uusi keskustelu' -nappia, aukeaa näkymä, jossa voi luoda uuden keskustelun määrittelemällä sille aiheen, viestin ja kuvan.

Loin näkymän, jolla voi luoda uuden keskustelun. Asetin modaalin stateen tiedon, ollaanko luomassa uutta keskustelua vai vastaamassa olemassa olevaan keskusteluun. Tätä käyttämällä pystyin renderöimään modaaliin halutun näkymän. Uuden keskustelun luontia varten tarvitsin tiedot keskustelun osallistujista. Raporttinäkymällä oli jo tieto käyttäjästä, jonka lomaketäyttöjä tarkasteltiin, joten sen pystyin antamaan käyttäjän id:n modaalille propseina. Kirjautuneen käyttäjän id sen sijaan löytyi Redux storesta. Lisäsin lähetettävään dataan listan keskustelun osallistujista ja uuden keskustelun aiheen. Sain siten uuden keskustelun luonnin toimimaan ongelmitta.

Tein vielä varsin monta pientä korjausta, refaktorointia tai lisäystä, mutta ihan kaikkea en saanut vielä valmiiksi. Viestimodaali alkoi kuitenkin olla valmis esiteltäväksi seuraavan viikon palaverissa.

Torstai 16.04.2020

Tavoitteenani oli viimeistellä viestimodaalin ensimmäinen versio. Kävin ulkoasusuunnitelman vielä läpi ja huomasin, että käyttäjälle pitäisi näyttää tietoja valitusta keskustelusta. Keskusteluiden otsikosta voi olla vaikeaa päätellä, onko kyseessä se keskustelu, johon haluaa lähettää viestin. Listaamalla keskustelun osallistujat ja viimeisimmän viestin päivämäärän, voimme helpottaa oikean keskustelun valintaa.

Teknisesti tämä oli helppo toteuttaa; tekemällä API-kutsun keskustelujen endpointiin valitun keskustelun id:llä, sain kaikki tarvitsemani tiedot, jotka sitten piti vain renderöidä keskusteluvalikon alle, kun keskustelu on valittuna. Keskustelun osapuolet olisivat yleensä raportteja luova käyttäjä ja valittu asiakas ja nämä olisivat suorituskyvyn kannalta nopeampaa hakea Redux storesta. Navisec Healthissa oli kuitenkin mahdollista luoda ryhmäkeskusteluja, joten osallistujien tiedot piti hakea tietokannasta.

Modaalista puuttui myös ilmoitus viestin onnistuneesta lähetyksestä tai aiheutuneesta virheestä. Tällä hetkellä näkymä vain nollautui viestin lähettämisen jälkeen. Normaalisti käytämme Redux storesta löytyviä notifikaatioita, jolloin pelkkä funktiokutsu halutulla viestillä lisää vihreän tai punaisen notifikaation sivun ylälaitaan. Modaalissa näitä ei kuitenkaan voinut käyttää, joten jouduin renderöimään notifikaatiot manuaalisesti. Lisäsin notifikaation tarvitsemat tiedot, kuten tekstin ja värin määrittelevän tyypin, modaalin stateen. Viestin lähettämisen jälkeen tarkistin API-kutsun statuksen, jonka perusteella pystyin asettamaan notifikaation tekstin ja tyypin. Näiden avulla pystyin lisäämään modaalin otsikon alle Notification -komponentin, joka ilmoittaa käyttäjälle viestin lähetyksen onnistuneen tai mahdollisesta lähetyksessä tapahtuneesta virheestä. Sain samalla idean, että voisin lisätä notifikaatioon linkin keskusteluun, johon viesti lähetettiin. Suora uudelleenohjaus viestin lähettämisen jälkeen ei tuntunut järkevältä, koska käyttäjä voisi haluta lähettää esimerkiksi useamman kaavion erillisinä peräkkäisinä viesteinä.

Linkki olemassa olevaan keskusteluun oli helppo luoda, koska valitun keskustelun ID löytyi statesta ja sen avulla pystyin luomaan URL:n keskusteluun. Uuden keskustelun kohdalla sen sijaan ID:tä ei ollut vielä olemassa, joten se piti ottaa viestin lähetyksen yhteydessä API-pyynnön lähettämästä vastauksesta:

```
// Lähetetään viesti back-endiin.
const response = await api.post(this.url, data)
if (response.ok) {
  // Jos pyyntö menee läpi, ja luomme uutta keskustelua, otetaan muuttujaan vastauksesta löytyvä ID.
  // Jos ei luoda uutta keskustelua, käytetään valitun keskustelun ID:tä.
  const threadId = this.state.newThread
    ? response.data.thread.id
    : this.state.selectedThread
  // Annetaan ID metodille, joka muodostaa siitä URL:n ja asettaa sen stateen.
  this.setThreadUrlToState(threadId)
}
```

Viestimodaali oli nyt valmis esiteltäväksi seuraavan viikon palaverissa. Työhön oli kulunut ehkä vähän enemmän aikaa kuin arvioin, mutta myös tekemistä oli huomattavasti enem-

män kuin mitä osasin ennakoida. Olin kuitenkin tyytyväinen siihen, mitä olin saanut aikaan. Vain kuvan lisääminen aiheutti normaalia suurempia ongelmia, muuten sain melkein kaiken tehtyä selvien suunnitelmien mukaisesti.

Seurantaviikko 7 Analyysi

Huonon edellisen viikon jälkeen tämä viikko tuntui hyvältä ja itseluottamus omaan tekemiseen kasvoi huomattavasti. Tavoitteenani oli saada viestimodaali valmiiksi ja työt etenivät joka päivä suunnitelmien mukaan. Sain lähes kaiken tehtyä ilman ylimääräistä tiedonhakua. Ehdin samalla panostaa siistiin koodiin ja hyvään käytettävyyteen. Tehtävät sujuivat niin hyvin, että ehdin työn ohessa tutkia myös JavaScriptin tyypitystä.

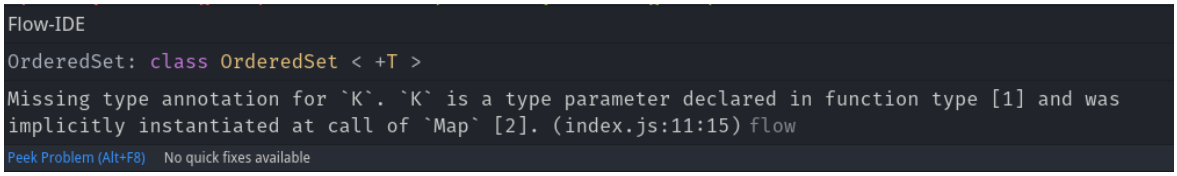
Olen viimeisten kuukausien aikana luonut paljon uusia React -komponentteja ja sitä myötä alkanut kiinnittää enemmän huomiota tyypitykseen. JavaScript on löyhästi tyypitetty kieli, joka tarkoittaa sitä, että käyttäjän ei tarvitse määrittää minkä tyyppistä informaatiota muuttujiin tallennetaan. (Yang, X. 2018) Staattisen tyypityksen avulla voimme nähdä virheellisten muuttujatyyppien aiheuttamat virheet jo koodieditorissa, joka nopeuttaa kehitysprosessia ja tekee koodin debugauksesta helpompaa.

Navisec Healthin kehityksessä käytämme JavaScriptin tyypittämiseen Facebookin kehittämää Flowta. Tiedän, että myös Microsoftin kehittämän TypeScriptin suosio on viime vuosina kasvanut suuresti, mutta itse en ole siihen vielä tutustunut, joten päätin selvittää, miten se vertautuu Flowhun. Olen aiemmin kuvitellut, että Typescriptin ja Flown merkittävin ero on se, että TypeScript on oma ohjelmointikielensä. Marius Schulz (2017) kuitenkin kirjoittaa TypeScriptin ja Flown eroavaisuuksia käsittelevässä blogikirjoituksessaan, että käyttipä TypeScriptiä tai Flowta, silloin ei kirjoita puhdasta JavaSciptia. Silloin ei käytetä kielen standardinmukaisia ominaisuuksia, joten sekä TypeScript, että Flow, voidaan käsittää omina kielinään. Syntaksiltaan molemmat kielet ovatkin hyvin samankaltaisia.

Samassa kirjoituksessaan Schulz (2017) määrittelee kielille kuitenkin selvän eron: TypeScript implementoi sekä tyyppitarkastajan, että kääntäjän, joka kääntää kielen puhtaaksi JavaScriptiksi. Flow tarkastaa tyyppejä ja tukeutuu Babeliin tai johonkin muuhun työkaluun, joka poistaa kielestä ylimääräiset tyyppimäärittelyt kieltä käännettäessä. (Schulz, M. 2017.) Yksi TypeScriptin suurimmista eduista tuntuisi olevan se, että se on paljon Flowta suositumpi. Niin ollen myös sen dokumentaatio on paljon laajempaa, ja apua ongelmiin on helppo löytää aktiivisesta käyttäjäkunnasta. Flown etuna on sen sijaan sen helppo integrointi olemassa oleviin projekteihin.

Omat kokemukseni flowsta ovat ristiriitaisia. Sen tuomia etuja on vaikea kiistää, mutta sen käyttöönotossa on ollut ongelmia tiimin jokaisella jäsenellä. Se on myös ainakin omalla työkoneellani raskas käyttää, jos koodieditoreita on auki enemmän kuin yksi. Staattinen tyypittäminen Flowlla vaatii jonkin verran työtä, sillä yhdessä React-komponentissakin voi olla paljon tyypitettävää. Metodien lisäksi tyypit määritellään mm. state-objektin propertyille sekä komponentille annettaville propseille. Näin ollen saamme koodieditorilta välittömästi palautetta, jos yritämme asettaa esimerkiksi boolean -arvoksi jotain muuta kuin *true* tai *false*.

Flow -huomautukset voivat kuitenkin olla hyvin epäselviä ja koodi saattaa kuitenkin toimia normaalisti, vaikka niitä ei korjaisi. Mahdollisesti tästä syystä monet komponentit sisältävät paljon korjaamattomia Flow-huomautuksia.



Kuva 6. Koodieditorissa käytettävän Flow-tulkin antama kryptinen virheilmoitus.

Itse en kuitenkaan koe Flow:n käyttöä järkevänä, jos sen tuomia ominaisuuksia ei hyödynnetä. Tämän takia olen pitänyt huolen siitä, että omiin komponentteihini ei Flow-huomautuksia jää ja olen samalla pyrkinyt korjaamaan niitä myös muista komponenteista.

3.8 Seurantaviikko 8

Tiistai 21.04.2020

Otin tavoitteekseni saada yksilöraportit valmiiksi tämän viikon aikana. Se tarkoitti ainakin viestimodaalin hyväksymistä tai viimeistelyä, Excel-exportin toteuttamista ja manuaalista testaamista. Aloitin aamuni testailemalla viestimodaalin toimintaa välttääkseni niin kutsutun "demoefektin" viikkopalaverissa eli odottamattomat ongelmat esityksen aikana. Ongelmia ei syntynyt ja palaute oli pelkästään positiivista. Sovimme myös, että kuvan lähettäminen nykyisellä tavalla oli riittävän helppo käyttökokemus eikä kehittyneempää versiota lähdettäisi ainakaan tässä vaiheessa tekemään. Päätimme, että modaaliin lisättäisiin vielä nappi, jolla pääsee aloitusnäkymään, jos on valinnut jonkin keskustelun tai aloittanut uu-

den. Tähän asti paluu on onnistunut vain sulkemalla modaalin ja avaamalla sen uudelleen.

Lisäsin modaalin *footeriin* napin, joka kutsui modaalin resetoivaa metodia, jota normaalisti käytetään viestin lähettämisen yhteydessä. Laitoin *footerin render* -metodiin myös ehtolausekkeen, joka renderöi napin vain silloin, jos oltiin näkymässä, josta piti palata takaisin alkuun.

Esittelyni aikana huomasin, että sekä raporttinäkymästä että viestimodaalista puuttui vielä käännöksiä. Käännöksien kanssa työskentely on usein erikoista. Front-endiin on luotu funktioita, jotka automaattisesti keräävät käännettävät tekstit *json* -tiedostoihin kaikilla kolmella eri kielellä (suomi, englanti, ruotsi), joita Navisec Healthissa käytetään. Teksti- ja nappielementtien sisällä olevien tekstien pitäisi mennä keräykseen automaattisesti, mutta tekstejä voi myös merkitä manuaalisesti:

```
// Tekstielementin teksti kerätään automaattisesti jokaisen kielen json-tiedostoon.
<Paragraph>
  Käännettävä teksti
</Paragraph>

// Tekstin voi myös merkata käännettäväksi, mutta sitä ei kerätä automaattisesti,
// vaan se pitää lisätä json-tiedostoihin itse.
<button>
  { __('Käännettävä teksti') }
</button>
```

Usein vastaan tulee kuitenkin lauseita, jotka sisältävät vaikkapa nimiä tai jotain muuta, mitä ei haluta lisätä käännöksiin. Tällöin elementille voidaan lisätä *notranslate -property*, tai antaa parametrinä vaikkapa sana, jota ei haluta kääntää:

```
<Text notranslate>
  Tätä tekstiä ei käännetä
</Text>

<Text params={{ paramName: 'parametri' }}>
  { 'Tämä ${paramName} jätetään kääntämättä, mutta kaikki muu käännetään.' }
</Text>
```

Pääasiassa nämä toimivat hyvin, mutta *JSON* -tiedostoihin kerääntyy usein paljon ylimääräistä tekstiä tai erikoismerkkejä ja niiden kääntäminen tai kääntämisen välttäminen vaatii välillä säätöä.

Olin tyytyväinen saamaani hyvään palautteeseen ja siihen, että yksilöraportit alkoivat kuukausien työn jälkeen olla valmiita testaukseen. Joutuisin vielä odottamaan design-tiimin näkemystä raporttinäkymän ulkoasusta, mutta tekemistäkin vielä riitti ainakin loppuviikon ajaksi.

Keskiviikko 22.04.2020

Päivän tavoitteenani oli edistää export -asioita, jotka olivat ainoa isompi yksilöraporteista puuttuva ominaisuus. Olin viime aikoina saanut pari ideaa PNG- ja PDF -exportien toteutukseen, mutta en ollut ehtinyt kokeilemaan niitä vielä. PNG-exportit pystyi kyllä ottamaan TUIChart -kaavion omasta export -valikosta, mutta valikko oli ulkoasultaan huono ja sisälsi enemmän export -vaihtoehtoja kuin mitä tarvitsimme.

Olin aiemmin ottanut html2canvas -kirjastolla PDF-tiedostoa varten kuvan kaikki kaaviot sisältävästä div-elementistä. Olin kokeillut myös ottaa kuvan yksittäisestä kaavion sisältävästä elementistä, mutta molemmat johtivat kuviin, jotka sisälsivät epämääräisiä harmaita alueita. Tällä kertaa loin TUIChart -kirjaston luoman kaavion ympärille tyhjän span-elementin, josta otin kuvan. Tällä kertaa sain haluamani kuvan otettua. Tämä tarkoitti sitä, että pystyin tekemään kaavion alle napin. Nappi kutsuu metodia, joka ottaa kaaviosta kuvan ja tallentaa sen käyttäjän koneelle. Lisäsin näkymään myös *Loader*-komponentin, jolla pystyi animoidun ikonin avulla indikoimaan, että näkymässä on lataus kesken. Kuvat kuitenkin latautuivat niin nopeasti, ettei ikonia ehtinyt edes näkemään.

Mietin pitkään, miten voisin hyödyntää kuvia PDF:n muodostamisessa. Kokeilin luoda ensin kaksi kuvaa html2canvas -kirjastolla ja lisätä ne PDF-tiedostoon *jsPdf* -kirjastolla. Lopputulos oli kuitenkin vain tyhjä sivu. Etsin internetistä tietoa ja sain selville, että kuvat luovat funktiot olivat asynkronisia, jonka takia PDF:n tallennus alkoi ennen kuin kuvat olivat valmiita. Lisäsin metodiin *async-await* -toiminnallisuuden ja sain luotua PDF:n, jossa oli toinen kuvista:

```
async generatePDF () {
  const pdf = new jsPdf('l', 'mm')
  // Valitaan ensimmäinen kaavioelementti ja otetaan siitä kuva
  const chart1 = document.getElementById('chart-0')
  await html2canvas(chart1, { scrollX: 0, scrollY: -window.scrollY })
    .then((canvas) => {
      let image1 = canvas.toDataURL('image/png')
    })
  // Kuva toisesta kaavioelementistä
  const chart2 = document.getElementById('chart-1')
  await html2canvas(chart2, { scrollX: 0, scrollY: -window.scrollY })
    .then((canvas) => {
      let image2 = canvas.toDataURL('image/png')
    })
  // Lisätään kuvat PDF-tiedostoon ja käynnistetään lataus
  pdf.addImage(image1, 'PNG', 0, 0, 210, 101)
  pdf.addImage(image2, 'PNG', 0, 0, 210, 101)
  pdf.save('report.pdf')
}
```

Tämä toimi niin kuin pitääkin. Pystyisin siis tekemään PDF:n, jossa jokaisella sivulla olisi yksi kaavio. Kokeilin luoda tällaisen PDF:n kääntämällä PDF:n orientaation vaakatasoon ja skaalaamalla kuvat noin yhden sivun kokoisiksi. Tämä toimi periaatteessa hyvin, mutta kuvien laatu alkoi heiketä skaalatessa.

En vielä löytänyt tyydyttävää ratkaisua PDF:n tekemiseen, mutta saavutin merkittäviä edistysaskeleita exportien kanssa.

Perjantai 24.04.2020

Olin vakuuttunut, että löytäisin tavan luoda PDF-exportin tämän päivän aikana. Päätin kuitenkin aloittaa päiväni käyttämällä kuukausittaiset opiskelutuntini Reactin opiskelemiseen. Olin jo pitkään halunnut opetella hookien käyttöä Reactissa, mutta olin tehnyt työn ohessa niin paljon Reactia, etten ollut vapaa-ajalla jaksanut panostaa tähän. Hookeja ei välttämättä tultaisi koskaan käyttämään Navisec Healthissa, mutta niiden opettelu tuntui tärkeältä oman kehityksen kannalta. Ostamallani kurssilla käsiteltiin *useState*- ja *useEffect* -hookien toimintaa. Molemmat vaikuttivat hyvin käteviltä ja kaksi tuntia meni todella nopeasti niiden käyttöä opetellessa.

Kun palasin PDF-exportien pariin, huomasin sattumalta, että kuvat eivät ylikirjoittuneetkaan niin kuin alun perin luulin, vaan dokumentissa oli kaksi kuvaa päällekkäin. Tämä oli

avainhuomio, sillä nyt minun tarvitsi vain saada kuvat sijoitettua oikeaan kohtaan dokumenttia. Kuvien sijoittelu oli helppoa, koska kuvalle sai määrittää x- ja y-koordinaatit. Olin jo aiemmin katsonut yhden kaavion korkeuden, joten asetin toisen kuvan kaavion korkeuden verran alemmaksi y-akselilla. Kuinka ollakaan, sain kaksi kaaviota näkyviin sivulle! Tyhjää tilaa jäi sen verran, että kolmatta kaaviota ei olisi enää saanut sivulle mahtumaan, mutta alas jäävä valkoinen alue oli silti vähän häiritsevä. Ratkaisin alas jäävän valkean alueen siirtämällä ensimmäistä kuvaa yläreunasta vähän alemmaksi ja lisäämällä kuvien välille vähän tyhjää tilaa.

Kahden kaavion sivu näytti hyvältä, joten minun piti enää keksiä, miten voisin luoda sen dynaamisesti välittämättä siitä, montako kaaviota raportilla on. Ensimmäinen ajatukseni oli käyttää *while-loopia*, jossa lisään jokaisen renderöidyn kaavion PDF-dokumenttiin. Tässä oli tärkeää käyttää nimenomaan renderöityjen kaavioiden lukumäärää mittarina, koska suodattimilla se voisi vaihdella. PDF:n tarkoituksena oli luoda tulostettava dokumentti näkymän sen hetkisestä tilasta, kun luontipainiketta painetaan. Loopiin piti myös luoda logiikka, joka jakaa kaaviot niin, että yhdelle sivulle tulee kaksi kaaviota, jonka jälkeen luodaan uusi sivu. Jouduin hieman pyörittelemään ehtolausekkeita ympäriinsä, mutta sain melko vaivattomasti aikaan toimivan loopin:

```
async generatePdf () {
  // Käytetään while-loopin laskurina renderöityjen kaavioiden lukumäärää
  const count = this.state.chartData.series.length
  let index = 0
  const pdf = new jsPdf('p', 'mm')
  while (index < count) {
    const element = `chart-${index}`
    const chart = document.getElementById(element)
    await html2canvas(chart, { scrollX: 0, scrollY: -window.scrollY })
      .then((canvas) => {
        const image = canvas.toDataURL('image/png')
        // Lisätään sivulle ensimmäinen kuva, ja siirretään sitä 10mm yläreunasta alaspäin
        if (index === 0 || index % 2 === 0) {
          pdf.addImage(image, 'PNG', 0, 10, 210, 101, element)
        }
        // Lisätään sivulle toinen kuva, ja siirretään sitä 145mm yläreunasta alaspäin
        // Jos kyseessä ei ole viimeinen kuva, lisätään dokumenttiin uusi sivu
        else {
          pdf.addImage(image, 'PNG', 0, 145, 210, 101, element)
          if (index !== count - 1)
            pdf.addPage()
        }
    })
    index += 1
  }
  pdf.save('report.pdf')
}
```

Olin hyvin tyytyväinen lopputulokseen. Sivulle jäi vielä jonkin verran tyhjää tilaa, mutta kaavioiden koon ja kuvasuhteen takia sille ei voinut oikein tehdä mitään. PDF:n luonti kesti jonkin aikaa, joten lisäsin vielä animoidun latausikonin näkymän päälle, kun raporttia luodaan.

Loppupäivän vietin kuukausittaisessa perjantaipalaverissa. Maanantaina olisi palaveri, jossa käydään vielä läpi yksilöraporttinäkymän ulkoasua. Ulkoasumuutosten jälkeen näkymästä puuttuisi enää Excel-exportit.

Viikko 8 Analyysi

Viikko meni nopeasti, pääasiassa PDF-tiedoston parissa. Viikot, jotka kuluvat yhtä pientä ominaisuutta säätäessä, tuntuvat usein siltä, ettei ole saanut mitään aikaiseksi. Uusia kirjastoja käyttäessä ne kuitenkin ovat usein hitaampia, kun dokumentaatiota pitää lukea, ja ongelmiin joutuu etsimään apua Googlesta normaalia enemmän. Löysin kuitenkin tällä viikolla ratkaisun jo useamman viikon auki olleeseen ongelmaan, joten hukkaan se ei missään tapauksessa mennyt. Joutuisin vielä säätämään tiedoston sisällön sijaintia sivulla, mutta nyt tiesin tarkalleen, miten se tapahtuisi. Se, että jouduin asentamaan kaksi kirjastoa lopputuloksen saavuttamiseksi ei ehkä ollut optimaalista, mutta en löytänyt ongelmaan muutakaan ratkaisua. Vastaavien ominaisuuksien koodaaminen alusta asti olisi todennäköisesti vienyt kuukausia, joten kirjastojen käyttö oli mielestäni perusteltua. Tämän myötä päätinkin tutustua Nodejs:n moduuleihin, ja niiden hallintaan käytettävään NPM (Node Package Manager) -paketinhallintajärjestelmään.

Paketinhallintajärjestelmän tarkoituksena on helpottaa Javascript-kehittäjien työskentelyä tarjoamalla helppokäyttöinen alusta avoimen lähdekoodin paketeille (Jämiä, M. 2019). Nykypäivänä paketteja löytyy niin paljon, että käytännössä minkä tahansa operaation voi suorittaa jonkin paketin avulla. Pakettien asentaminen NPM:in avulla on myöskin helppoa. Paketin voi asentaa suoraan komentorivin kautta, ja se lisätään automaattisesti *package.json* -nimiseen tiedostoon. Kaikki package.json -tiedostossa olevat paketit voidaan asentaa myöskin yhdellä komennolla, joten sovellus on helppo siirtää koneelta tai palvelimelta toiselle. Miksi kaikkea ei sitten tehdä valmiiden pakettien avulla, vaan koodataan käsin?

Julie Ng (2018) listaa Nodejs -moduuleja koskevassa kirjoituksessaan kolme niiden aiheuttamaa turhautumisen kohdetta:

Nodejs -moduulit sisältävä *node_modules* kansio voi kasvaa kooltaan suureksi, ja sisältää tuhansia tiedostoja.
Moduulit eivät tallennu "globaaliin" välimuistiin.
Asentaaksesi moduuleita komentoriviltä, tarvitset aina internetyhteyden.
Näistä etenkin ensimmäinen kohta on tullut itsellenikin tutuksi omista projekteista. Esimerkiksi Facebookin luoma *create-react-app* -moduuli auttaa käynnistämään uuden React-projektin tyhjästä. Se asentaa kaikki tärkeimmät moduulit kuten itse Reactin, ja monia siihen liittyviä työkaluja kuten yksikkötestausmoduulin ja TypeScript -tuen. Tämän seurauksena kuitenkin vasta-aloitetun React-projektin koko on jo alkupisteessä useita satoja megatavuja. Näiden ongelmien lisäksi itselleni tuli mieleen myös tietoturva-asiat. Tietoturvahaavoittuvuudet ovat suuri ongelma avoimen lähdekoodin Node.js-paketeissa. Haavoittuneet paketit voivat pahimmassa tapauksessa avata sovelluksen erilaisille hyökkäyksille. (Jämiä, M. 2019)

Nodejs -moduulit ovat olleet suuri apu omissa projekteissani, ja aion jatkaa niiden käyttöä niin henkilökohtaisissa projekteissa, kuin työssäni. Olen kuitenkin oppinut punnitsemaan niiden tarpeellisuutta. Harkitsen tapauskohtaisesti, olisiko jokin ominaisuus järkevämpää toteuttaa itse, vai kannattaisiko siihen käyttää moduulia. Jos päädyn käyttämään moduulia, pyrin valitsemaan moduulin, joka on mielellään pieni kooltaan, eikä sisällä tarpeettomia moduuliriippuvaisuuksia. Pyrin myös auditoimaan moduulit tasaisin väliajoin NPM:in avulla, jotta niiden tietoturva säilyy parhaalla mahdollisella tasolla.

3.9 Seurantaviikko 9

Maanantai 27.04.2020

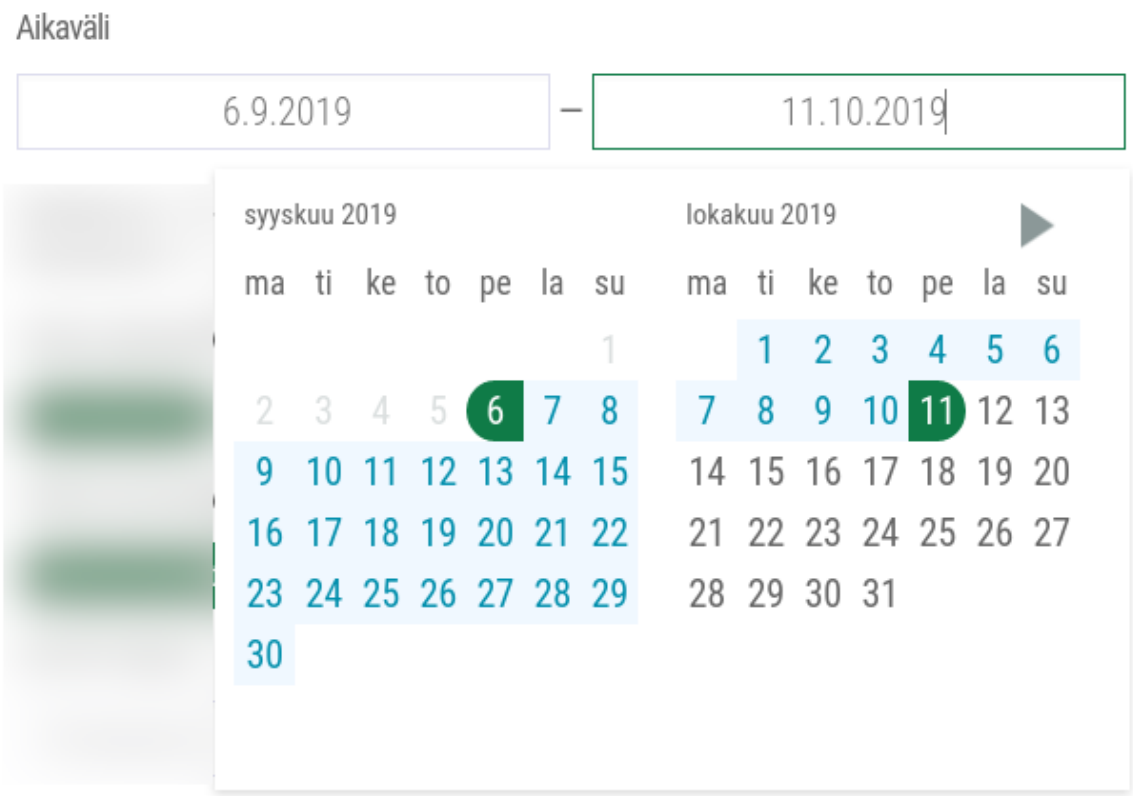
Tavoitteenani oli saada tämän viikon aikana valmiiksi palaverissa sovittavat ulkoasumuutokset ja Excel-export-toiminnallisuus. Jos näihin ei kuluisi koko viikkoa, voisin vielä käyttää aikaa manuaaliseen testaamiseen. Mikäli ongelmia ei tulisi vastaan, yksilöraportoinnin ensimmäinen versio voitaisiin laittaa testipalvelimelle laajempaan testaukseen.

Pidimme aamulla palaverin Design-tiimin kanssa mahdollisista ulkoasumuutoksista. Kävimme läpi koko raportointinäkymän ja päätimme, että muuttaisimme raportin suodatinpainikkeiden ulkoasua ja raportin päivämääräsuodattimien komponenttia. Kävimme palaverin jälkeen vielä yhdessä tuoteomistajamme kanssa komponenttivaihtoehtoja päivämääräsuodattimille. Käytimme tällä hetkellä kahta react-day-picker -kirjaston komponenttia

sekä alku- että loppupäivämäärän valintaan. Kävin läpi kirjaston muita esimerkkejä, ja ehdotin toiminnallisuutta, jossa sekä kentät että niistä aukeavat kalenterit olisi yhdistetty samaan toiminnallisuuteen. Näin välttyisimme uuden kirjaston asentamiselta ja saisimme paremman näköisen komponentin, joka myös parantaa käyttökokemusta. Ehdotukseni hyväksyttiin ja pian sainkin ulkoasusuunnitelman uusista suodatinpainikkeista.

Suodatinpainikkeiden vaihto oli varsin yksinkertaista. Korvasin PrimaryButton -painikkeet Badge -komponenteilla. Badgeille ei kuitenkaan ollut rakennettu mitään painikeominaisuuksia, vaan niitä oli käytetty vain staattisina ulkoasuelementteinä. Koska suodattimissa käytettävät painikkeet olivat kytkettäviä painikkeita (on/off), ollen niin erikoistapauksia, päätin luoda toiminnallisuuden suoraan raporttinäkymään. Loin CSS-luokat, jotka vaihtuvat, kun nappia painetaan. Näin napin ulkoasu indikoi, onko se päällä vai ei. Jouduin vielä jonkin verran säätämään tyylityksiä, mutta sain varsin nopeasti luotua ulkoasusuunnitelman mukaiset painikkeet.

Uuden kalenteritoiminnallisuuden arvelin olevan monimutkaisempi tehtävä. Aloitin luomalla uuden komponentin, johon kopioin nykyisen kalenteritoiminnallisuuden. Sen jälkeen tein niin kuin usein ulkopuolisten kirjastojen kanssa olen tehnyt, kopioin kirjaston dokumentaatiosta löytyvän esimerkin suoraan komponentin sisään ja kokeilin mitä tapahtuu. Lisäsin myös kalenterin vaatimat tyylitykset omaan tiedostoonsa ja renderöin komponentin näkymään. Komponentti toimi yllättävän hyvin ilman kustomointia. Päivämäärien valitseminen onnistui ja tuplakalenterit näyttivät hyvältä. Lokalisaatiotoiminnallisuudet piti kuitenkin vielä siirtää aikaisemmasta komponentista uuteen komponenttiin, sillä päivämäärät olivat Yhdysvaltain formaatissa ja kalenterit englanninkielisiä. Kopioin lokalisaatiologiikan vanhasta komponentista ja tein vielä manuaaliset käännökset päivämääräkenttien placeholdereille.



Kuva 7. Kuva uudesta kalenterikomponentista, jolla käyttäjä voi valita raportin aikavälin.

Kalenteri ei vielä toiminut, jos päivämäärän kirjoitti manuaalisesti kumpaan tahansa kenttään, mutta en uskonut tämän haittaavan, ainakaan tässä vaiheessa. Olin kuitenkin positiivisen yllättynyt, kuinka helposti sain uuden kalenterikomponentin tehtyä.

Tiistai 28.04.2020

Päivän tavoitteenani oli viimeistellä edellisen päivän muutokset ja palata Excel-exportin pariin, jos viikkopalaverissa ei tulisi mitään yllätyksiä.

Esittelin tekemäni muutokset viikkopalaverissa ja sain hyvää palautetta molemmista. Lupailin myös, että yksilöraporttinäkymä olisi valmis testattavaksi seuraavalla viikolla. Yhteiset resurssimme kuitenkin vähenivät taas viikon vaihteessa, kun yksi tiimiläisistämme lopetti. Olisin siis pitkästä aikaa vastuussa myös raporttinäkymän ulkopuolisista bugikorjauksista.

Kävin vielä kalenterikomponenttia läpi ja refaktoroin metodeja vähän lyhyemmiksi. Huomasin, että olin unohtanut merkitä tiedoston Flow-tagilla, joten tiedostosta puuttui vielä tyypitys kokonaan. Samalla kun lisäsin Flow-tagin, lisäsin komponentille lyhyen dokumen-

taation, jossa kerrotaan komponentin käyttötarkoitus ja ohjeet sen lisäämiseen johonkin näkymään. Lisäsin muuttujille ja metodeille tyypit ja korjasin Flown ilmoittamat varoitukset. Kun komponentti oli mielestäni siisti, dokumentoitu ja varoituksia ei koodieditorissa näkynyt, lähetin muutokseni GitLabiin pieninä commiteina.

Loppupäivän yritin saada Excel-exportia toimimaan, mutta törmäsin samoihin ongelmiin kuin aiemmin; sain tiedoston ladattua, mutta sen sisältö oli korruptoitunut. Ymmärtääkseni ongelma oli omassa http-rajapinnassamme. En saanut määriteltyä vastauksen tyyppiä, jolla tiedoston sisältö olisi tulostunut oikein vastaukseen. Vastauksen tyyppi olisi helppo määritellä esimerkiksi suositulla axios -kirjastolla, mutta axiosilla tehdyt pyynnöt eivät menisi läpi autentikaation takia.

Excelin lataaminen vaikutti edelleenkin ongelmalta, jonka pitäisi olla helposti ratkottavissa, mutta en tälläkään kertaa juuri edistynyt sen kanssa. Uskoin kuitenkin olevani lähellä ratkaisua, joten en antanut asian haitata liikoja, kun työviikkoakin oli vielä yksi päivä jäljellä.

Torstai 30.05.2020

Aloitin päiväni tutustumalla omaan http-rajapintaamme. Oma dokumentaatiomme sille oli suppeaa, mutta löysin sieltä tiedon, että se on rakennettu apisauce -nimisen kirjaston päälle. Voisin siis tutkia apisaucen dokumentaatiota selvittääkseni, miten voisin vaihtaa vastauksen tyyppiä.

Apisaucen dokumentaatio oli kattavaa ja helppolukuista, mutta sieltä löytyvien esimerkkien soveltaminen omaan kustomoituun rajapintaamme ei ollut niin suoraviivaista kuin toivoin. Lopulta sain muutettua vastauksen tyyppiä ainakin nimellisesti, mutta vastauksen data oli edelleen korruptoitunutta. Vastauksen tyypin muuttaminen ei tuntunut tekevän mitään. Yritin vielä muodostaa tiedostoa eri tavalla back-endissä, mutta lopputulos oli aina sama.

API-ongelma alkoi turhauttaa toden teolla, joten otin iltapäivällä ilolla vastaan bugikorjauksen, vaikka se tarkoitti sitä, että joutuisin vaihtamaan Git branchia. Branchin vaihtaminen tarkoitti sitä, että joutuisin ajamaan paljon migraatioita. Se vuorostaan tarkoitti sitä, että palatessani raporttien branchiin, joutuisin ajamaan migraatioita takaisin päin.

Bugin korjaus osoittautui helpoksi tehtäväksi. Uusien käyttäjien rekisteröinti omaan hallintaosioomme oli rikki, koska käyttäjät rekisteröitiin käyttöliittymässä pelkän sähköpostiosoit-

teen avulla, mutta back-end odotti myös käyttäjänimeä. En kuitenkaan voinut poistaa käyttäjänimeä back-endistä, sillä sitä käytettiin sisäänkirjautumiseen. Korjasin siis ongelman lisäämällä sähköpostiosoitteen myös käyttäjänimeksi front-endissä pyyntöä lähetettäessä.

Seurantaviikko 9 Analyysi

En mitenkään voinut olla tyytyväinen viikon saavutuksiini, vaikka koin kyllä yrittäneeni kaikkea mitä osasin. En vieläkään voinut uskoa, että simppelin Excel-tiedoston tekeminen ja lataaminen voisi koitua näin suureksi ongelmaksi. En ollut kuitenkaan tutustunut aikaisemmin omaan http-rajapintaamme sen tarkemmin, joten opin kyllä paljon uutta sen toiminnasta. Olen myös viime aikoina huomannut kehittyneeni dokumentaatioiden lukemisessa. Olen kyllä käyttänyt useita avoimia rajapintoja aikaisemminkin, mutta halusin tutustua rajapintoihin tarkemmin.

Rajapintojen avulla ohjelmistot voivat olla yhteydessä muihin ohjelmistoihin ja kommunikoida keskenään, riippumatta siitä, millä ohjelmointikielillä ja teknologioilla rajapinnat on kehitetty. (Aalto, O. 2019). Monet rajapinnat ovat pelkkiä datarajapintoja, jotka palauttavat kutsuttaessa dataa yleensä XML- tai JSON-muodossa. Tällaisia rajapintoja olen käyttänyt useasti omissa projekteissani ja jopa koulutöissäni. Navisec Healthissa käytämme REST-arkkitehtuurimallia, johon myös monet moderneista web-rajapinnoista perustuvat. REST-arkkitehtuurimallissa resursseja käsitellään http-protokollan tarjoamilla metodeilla, joista käytetyimmät ovat GET, POST, DELETE, UPDATE ja PATCH (Kivisaari, T. 2016).

JavaScriptillä http-protokollan metodeja on perinteisesti tehty AJAX:n avulla, mutta etenkin Nodejs:n yleistyttyä, tilalle on tullut erilaisia kirjastoja. Kuten viikkoraportissani kirjoitin, itse käytämme apisauce -nimistä kirjastoa, jonka päälle olemme rakentaneet omaa toiminnallisuutta. Omat toiminnallisuudet mahdollistavat esimerkiksi juuri oikeanlaisen autentikoinnin. Voisimme käyttää muita kirjastoja ulkopuolisten rajapintojen kutsumiseen, mutta omaa rajapintaamme voimme kutsua vain omalla tekniikallamme. Lukemalla apisaucen dokumentaatiota pääsin kuitenkin suhteellisen helposti sisään luomiimme toiminnallisuuksiin, vaikka lopulta mitään muutoksia ei tällä kertaa tarvinnutkaan tehdä.

Rajapinnat ja etenkin avoimet rajapinnat ovat mielestäni mullistaneet web-kehitystä viimeisen kymmenen vuoden aikana. Aloittelevatkin ohjelmoijat pystyvät helposti luomaan mielenkiintoisia web-sovelluksia, joissa hyödynnetään avointa dataa.

3.10 Seurantaviikko 10

Tiistai 05.05.2020

Viimeisen seurantaviikkoni tavoitteena oli saada koko seuranta-ajan tekemäni yksilöraporttiominaisuuden ensimmäinen versio testikelpoiseen kuntoon. Se tarkoitti enää Excel-exportin toteuttamista ja muutamia pienempiä viilauksia. Toivoin saavani exportin vihdoinkin valmiiksi tämän päivän aikana, mutta aikaisempien ongelmien takia tiesin, että siihen saattaisi mennä vielä useampi päivä.

Viikkopalaverissa kerroin olevani lähellä ratkaisun löytämistä, ja lupailin, että tällä viikolla saisimme ominaisuuden ainakin oman tiimimme sisäiseen testiin. Sain myös ohjeistuksen, että CSV-tiedosto riittäisi ensimmäiseen versioon. CSV-tiedosto (Comma Separated Value) sisältää vain pilkulla eroteltuja arvoja, joten se on yhteensopiva monen eri ohjelman kanssa. CSV-tiedostossa ei voi hyödyntää samoja ominaisuuksia kuin Excelin omissa XLS- tai XLSX-tiedostoissa, mutta asiakas pystyisi kuitenkin lataamaan lomakeraportin tiedot taulukkomuodossa.

CSV-tiedoston luonti Djangolla ei myöskään vaadi ulkopuolisia kirjastoja, vaan sen voi luoda Pythonin omalla csv -moduulilla. Vaihdoin näkymäni luomaan CSV-tiedoston, jonka ensimmäisellä rivillä on satunnaisia sanoja testidataa. Latasin tiedoston, ja huomasin, että se toimi niin kuin pitääkin. Pystyin poistamaan edellisenä päivänä tekemäni API-kokeilut ja tiedosto latautui silti oikein.

Vaikka parin päivän työ tuntui taas menneen hukkaan, olin ilahtunut siitä, että ainakin tämä toteutus toimi niin kuin halusin. Pian kävi kuitenkin selväksi, että oikean datan saaminen CSV-tiedostoon ei olisikaan niin helppoa, kuin voisi kuvitella. Taulukkoon olisi helppoa luoda rivejä lataamalla niihin suoraan dataa Djangon modeleista, mutta tarvitsemalleni datalle ei ollut mitään yksittäistä modelia. Sen sijaan joutuisin yhdistelemään ja suodattamaan tarvitsemani datan tietokannasta, ja lopuksi muotoilemaan sen oikeaan muotoon.

Käytin iltapäiväni datan suodattamiseen ja muotoiluun, mutta en juuri edennyt sen saamisessa taulukkomuotoon. Päätin jatkaa CSV-tiedoston luomista seuraavana päivänä.

Keskiviikko 06.05.2020

Keksin edellisenä iltana vaihtoehtoisen tavan luoda CSV-tiedosto Djangolla. Sen sijaan, että tekisin monimutkaista suodatusta ja datan muotoilua back-endissa, voisin luoda tarvittavat rivit Reactissa, ja lähettää ne back-endiin POST-pyynnöllä. Front-endissä data oli jo valmiiksi paremmassa muodossa, joten datarivien luominen oli helppoa. Back-endin viewin tarvitsi siis vain poimia datarivit pyynnön parametreista, kirjoittaa ne CSV-tiedostoon, ja palauttaa ladattava tiedosto front-endille.

Kaikki tämä oli varsin helppoa toteuttaa, ja lopputuloksena oli juuri sellainen tiedosto, kuin halusinkin. Pienen testailun jälkeen kuitenkin huomasin, että jos raportin dataa suodatti, niin CSV:n sisältämät rivit menivät sekaisin. Olin epähuomissa käyttänyt suodatettavaa dataa rivien luontiin. Näkymän statesta löytyi myös datasetti, jonka ei pitäisi muuttua muulloin, kuin lomaketta tai suodatuksen päivämääriä muuttaessa. Ongelma kuitenkin säilyi, vaikka vaihdoin datasettiä. Dataa ei muutettu missään millään tavalla, mutta silti se tuntui muuttuvan, kun printtasin sen tietoja selaimen konsoliin. En ehtinyt kauaa tutkia asiaa, ennen kuin harjoittelijamme pyysi minulta apua omaan ongelmaansa. Loppupäivän autoin häntä etäyhteyden avulla.

Olin ärsyttävän lähellä ratkaisua, mutta taas uusi outo ongelma söi motivaatiota.

Torstai 07.05.2020

Raportointiominaisuutta haluttiin testattavaksi niin kovasti, että päätimme, että osittain toimiva CSV-export olisi tässä vaiheessa riittävä. Voisin kokeilla testipalvelimen päivitystä ensimmäistä kertaa itse. Kävin kuitenkin läpi vielä koko näkymän ja korjasin pari pientä bugia, jotka olivat jääneet roikkumaan.

Testipalvelimen päivittämiseen oli olemassa ohjeet, mutta niistä puuttui pari melko oleellista vaihetta. Sain kuitenkin tiiminvetäjältämme apua pienen odottelun jälkeen. Päätimme myös, että raportointinäkymä pitäisi näkyä testipalvelimella, mutta se ei saisi näkyä tuotannossa seuraavan päivityksen ohessa. Kaikki ominaisuudet kuitenkin kulkevat testipalvelimen kautta tuotantoon. Meillä ei ollut mitään valmista toiminnallisuutta, jolla voisi määritellä, että jokin osa näkyisi testillä, mutta ei tuotannossa. Päädyimme käyttämään Dockerin ympäristömuuttujia, sillä niillä voisi määritellä omia muuttujia kehitys-, testi- ja tuotantoympäristöihin.

Dockerin ympäristömuuttujat olivat itselleni uusi asia, joten jouduin jonkin verran tutkimaan, miten ne toimivat. En saanut testattua niitä paikallisessa kehitysympäristössäni,

joten jouduin päivittämään testipalvelimen useaan otteeseen. Päivittäminen oli helppoa, mutta siihen kului kuitenkin aikaa.

Juuri seurantajaksoni viimeisen työpäivän päätteeksi sain julkaistua yksilöraportointiominaisuuden testipalvelimellamme. Se ei ollut vielä valmis, ja korjauksiakin tulisi varmasti eteen, kun sitä testaisi useampi henkilö, tuntui se silti onnistuneelta päätökseltä kymmenen viikon seurantajaksolle.

Seurantaviikko 10 Analyysi

Yksilöraportointiominaisuuden saaminen testipalvelimelle juuri seurantajakson loppuessa oli suuri helpotus. Viimeiset viikot alkoivat olla varsin hektisiä, kun raportointiosion viimeistelyn ja päiväkirjaraportoinnin ohessa jouduin vielä korjailemaan bugeja ja avustamaan harjoittelijaamme. Tein myös ensimmäistä kertaa työssäni Git mergejä, joissa yhdistelin kokonaisia brancheja toisiinsa. Tässä ei kuitenkaan ollut mitään muuta erityistä, kuin varsin paljon merge konflikteja, koska brancheja ei oltu mergetty moneen kuukauteen. Tässä olikin hyvä oppi tulevaisuuteen; pääkehitysbranch kannattaa mergetä isoille bugikorjauksille tai uusille ominaisuuksille luoduille brancheille tasaisin väliajoin, jotta konflikteja ei pääse kertymään. Jouduin viimeisenä päivänäni tutustumaan tarkemmin myös Dockeriin ja sen mahdollistamaan automatisoituun julkaisuun testipalvelinta päivittäessäni, joten päätin viikon päätteeksi perehtyä Dockeriin vielä tarkemmin.

> *Docker on työkalu, jonka tehtävä on helpottaa sovellusten luomista, julkaisua ja pyörittämistä konttien avulla. Kontit mahdollistavat sovelluksen pakkaamisen yhteen pakettiin, joka sisältää kaikki sovelluksen tarvitsevat osat, kuten kirjastot ja muut riippuvuudet. (Simonov, O. 2019.)*

Kontit toimivat siis aina samalla tavalla, vaikka niiden ympäristö vaihtuisi kehittäjän tietokoneelta testi- tai tuotantoympäristöön. Eri ympäristöissä konttien toimintaa voidaan kustomoida konfiguraatiotiedostojen avulla. Docker -kontin luomiseen tarvitaan normaalisti Docker -tiedosto (Dockerfile), joka on sovelluksen pohjatasolla sijaitseva tekstitiedosto. Docker -tiedostossa määritellään riveittäin komentoja, joilla määritellään muun muassa sovelluksen käyttämä image (esim. Nodejs) ja paljastetaan sovellukselle portti. Docker suorittaa komennot rivi kerrallaan, joten jos imageksi on valittu Nodejs, voidaan portin paljastamisen jälkeen ajaa komento "npm start", joka käynnistää Nodejs -palvelimen.

Useat kontit voidaan sen sijaan yhdistää Docker Composen avulla. Docker Compose antaa kehittäjille mahdollisuuden luoda YAML-konfiguraatiotiedosto, jolla sovelluspalvelu voidaan käynnistää yhdellä komennolla (Tanner, G.). Docker Composella voidaan siis luoda yhtä komentoa käyttämällä kokonainen kehitysympäristö front-endeineen, back-endeineen ja tietokantoineen.

Gabriel Tannerin mukaan Docker Composen konfiguraatiotiedoston tulee sisältää ainakin seuraavat neljä asiaa:

- Compose -tiedoston versio
- Kaikki rakennettavat palvelut
- Kaikki käytettävät volyymit
- Kaikki tietoverkot, jotka ovat yhteydessä eri palveluihin.

Olin kokeillut Dockeria ennen nykyisen työni aloittamista, mutta Docker Compose oli minulle kokonaan uusi tuttavuus. Käytän sitä kyllä periaattessa päivittäin työssäni, mutta tavallisesti käytän vain muutamaa peruskomentoa, joilla esimerkiksi ajan kehitysympäristöni ylös tai alas. Viimeisellä viikollani jouduin kuitenkin tutustumaan Docker Composen YAML-konfiguraatiotiedostoihin, joten vaikka Dockerissa riittää opiskeltavaa pitkäksi aikaa, koen oppineeni varsin paljon Dockerin ja Docker Composen perustoiminnasta viimeisen viikon aikana.

Omat kokemukseni Dockerista ja Docker Composesta ovat olleet hyvin positiivisia. Kokonaisen kehitysympäristön pystyttäminen yhdellä komennolla niin omalla tietokoneella kuin testipalvelimellakin, on uskomattoman helppoa ja nopeaa. Alkuun Docker Composen peruskäyttökin vaati totuttelua, sillä komentoja oli helppo suorittaa väärissä kansioissa ja virheviestit saattoivat olla vaikeasti tulkittavia. Sittemmin olen kuitenkin vakuuttunut sen tuomista hyödyistä ja aion ehdottomasti tulevaisuudessa käyttää Dockeria ja Docker Composea omissakin projekteissani.

4 Pohdinta ja päätelmät

Kymmenen viikon seurantajaksoni oli monella tapaa poikkeuksellinen. Itse raportoinnin ohella jouduin totuttelemaan kokoaikaiseen etätyön tekemiseen, ja työtehtävät olivat aiempiin kuukausiin verrattuna yksitoikkoisempia. Toisaalta yhden ominaisuuden työstäminen koko seurantajakson ajan helpotti raportointia, mutta toisaalta en tehnyt koko jakson aikana juuri ollenkaan esimerkiksi back-end-puolen hommia Djangolla.

Juuri seurantajakson lopussa minulla tuli yksi vuosi täyteen FNS:lla. Vuosi on mennyt todella nopeasti, ja ensimmäiset hermostuneet viikot web-kehittäjänä uuden projektin parissa ovat vielä hyvin muistissa. Yksi suurimmista havainnoistani raportointijakson aikana olikin se, kuinka vaikeaa omaa oppimista on lyhyellä aikavälillä havainnoida, etenkin ensimmäisiin kuukausiin verrattuna. Etenkin viikkotasolla omaa kehittymistä oli vaikeaa analysoida, kun työskentely oli jo varsin rutiininomaista. Päiväkirjan kirjoittaminen, ja etenkin viikkoanalyysit kuitenkin auttoivat hahmottamaan omaan kehitykseen vaikuttaneita seikkoja.

Vietin lähes koko seurantajaksoni Reactin parissa, ja sen parissa olen kokenut suurimmat edistysaskeleeni. Olen oppinut luomaan kokonaisia komponentteja modernia JavaScript-syntaksia käyttäen. En enää tyydy ensimmäiseen ratkaisuun, joka toimii niin kuin on haluttu, vaan käyn läpi myös muita mahdollisia ratkaisuita. Jos parempia ratkaisuita ei löydy, pyrin pitämään huolen, että oman ratkaisuni logiikka on siistiä, eikä aiheuta ei-toivottuja sivuvaikutuksia. Olen myös oppinut käyttämään moduuleja harkitummin, ja pyrin valitsemaan turvallisia, ja vähän riippuvuuksia sisältäviä moduuleja. Lisäksi olen oppinut tyypittämään koodini, ja osaan nykyään korjata melkein kaikki Flow-huomautukset koodistani.

Olen oppinut dokumentoimaan koodiani niin front- kuin back-endissä. Jouduin itse alussa oppimaan paljon sovelluksen toiminnasta ja komponenteista ilman dokumentaatiota, ja tämä on toiminut suurena motivaattorina dokumentaation luontiin. En ehkä osaa vielä tunnistaa milloin dokumentaatio on oikeasti tarpeellista, mutta uskon, että olen kehittynyt siinäkin huomattavasti, ja tulen kehittymään tulevaisuudessakin.

Yksi parhaista tilaisuuksista havaita omaa kehittymistä on ollut uuden työntekijän auttaminen. Auttamisessa on ollut omat haasteensa, etenkin etätyöskennellessä, mutta oma ymmärrys sovelluksesta ja etenkin sen React-komponenteista on parantunut valtavasti. Pystyn usein tunnistamaan ongelmakohdat, ja osaan käyttää tarjolla olevia työkaluja kuten selaimen kehittäjätyökaluja ongelman selvittämiseen. Samalla olen kuitenkin huoman-

nut, että en itse osaa vieläkään pyytää apua tarpeeksi usein. Käytän toisinaan turhan paljon aikaa jonkin ongelman selvittämiseen, kun voisin saada ratkaisun nopeasti joltain muulta tiimiläiseltä. Tähän on vaikuttanut todennäköisesti etätyöskentely ja se, että viimeiset kuukaudet ovat olleet koko tiimille kiireellisiä.

Olen myös oppinut lisää muista projektiin liittyvistä teknologioista, kuten Dockerista. Päiväkirja-analyyseihin tehdystä tutkimustyöstä oli etenkin tässä paljon apua. Pelkästään Dockerin opetteluun voisi käyttää kuukausia, mutta nykyiseen työskentelyyni se olisi tarpeetonta. Perustoiminnallisuuksista on kuitenkin hyvä olla perillä, vaikka jokapäiväinen käyttö perustuisikin vain muutaman komennon käyttöön.

Oppimisprosessi ei kehittäjän työssä lopu koskaan. Teknologia kehittyy kovaa vauhtia, ja muutoksia työympäristöön tulee jatkuvasti. Ensimmäisten kuukausien aikana olin varma, etten selviäisi harjoittelujaksoa pidemmälle. Myöhemmin itseluottamus omaan tekemiseen on kuitenkin kasvanut, ja vastuuta olen saanut sen mukaisesti. Vaikka päiväkirjan kirjoittaminen normaalin työskentelyn ohessa oli kuormittavaa ja usein vaikeaa, on se auttanut minua huomaamaan, että kehityn jatkuvasti. Olen kokenut itseni onnekkaaksi, kun voin työskennellä projektissa, jossa kehitetään elinkaaren alkuvaiheessa olevaa modernia sovellusta, juuri niillä tekniikoilla, mistä olen kiinnostunut. Olen myös sopeutunut etätyön tekoon, vaikka se onkin vähentänyt sosiaaliset kontaktit minimiin. En ole tehnyt pitkän tähtäimen suunnitelma, mutta toivon voivani kehittää taitojani näissä töissä vielä pitkään. React-osaamisen lisäksi haluan oppia myös paremmaksi back-end-kehittäjäksi, ja oppia tuntemaan erilaisia teknologioita paremmin.

Lähteet

Onatsu, J. 2018. *Ohjelmointikäytännöt ja Dokumentointi*. Savonia ammattikorkeakoulu, tekniikan ja liikenteen ala. Kuopio. Luettavissa: https://www.theseus.fi/bitstream/handle/10024/153574/Onatsu_Joonas.pdf?sequence=1&isAllowed=y. Luettu 8.3.2020.

Freeman, A. 2019. *Pro React 16*. Apress. Luettavissa: https://learning.oreilly.com/library/view/pro-react-16/9781484244517/. Luettu 8.3.2020.

Gera, S. 2017 *How to Get Better at Estimating Software Development Time*. Hacker Noon. Luettavissa: https://hackernoon.com/barriers-to-effective-software-effort-estimation-and-how-to-avoid-them-4abd39f09f26. Luettu 6.4.2020.

Fournova Software GmbH. *Learn Version Control with Git*. Luettavissa: https://www.git-tower.com/learn/git/ebook/en/command-line/introduction. Luettu 6.4.2020.

Hunt, P. 2013. *Why did we build React?*. React Blog. Luettavissa: https://reactjs.org/blog/2013/06/05/why-react.html. Luettu 9.4.2020.

Freitas, V. 2016. *How to Use Django's Generic Relations*. simple is better than complex. Luettavissa: https://simpleisbetterthancomplex.com/tutorial/2016/10/13/how-to-use-generic-relations.html. Luettu: 9.4.2020.

Dodds, K. 2019. *Write tests. Not too many. Mostly integration*. Kent C. Dodds Blog. Luettavissa: https://kentcdodds.com/blog/write-tests. Luettu: 13.4.2020.

Boér, M. 2019. *Generating PDF from HTML with Node.js and Puppeteer*. RisingStack Blog. Luettavissa: https://blog.risingstack.com/pdf-from-html-node-js-puppeteer/. Luettu 13.4.2020.

Schulz, M. 2017. *TypeScript vs. Flow*. Marius Schulz Blog. Luettavissa: https://mariusschulz.com/blog/typescript-vs-flow. Luettu 24.4.2020.

Pressman, R. 2010. *Software Engineering: A Practitioner's Approach (7th edition)*. McGraw-Hill Higher Education. Luettavissa: http://seu1.org/files/level4/IT-242/Software%20Engineering%20_%207th%20Edition.pdf. Luettu 2.5.2020.

Jämiä, M. 2019. *Tietoturvaongelmien havaitseminen Node.js-ympäristöissä*. Metropolia ammattikorkeakoulu, tieto- ja viestintätekniikka. Luettavissa: https://www.theseus.fi/bitstream/handle/10024/168158/Ja%CC%88mia%CC%88_Mikko.pdf?sequence=2&isAllowed=y. Luettu 9.5.2020.

Ng, J. 2018. *JavaScript Best Practices and node_modules*. Julie Ng Blog. Luettavissa: https://julie.io/writing/javascript-node-modules/. Luettu 9.5.2020.

Aalto, O. 2019. *REST-ARKKITEHTUURIN HYÖDYNTÄMINEN WEBRAJAPINNAN KEHITTÄMISESSÄ*. Satakunnan ammattikorkeakoulu, tieto- ja viestintätekniikan koulutusohjelma. Luettavissa: https://www.theseus.fi/bitstream/handle/10024/166723/Opinn%c3%a4ytety%c3%b6_Olli_Aalto.pdf?sequence=2&isAllowed=y. Luettu 13.5.2020.

Kiviniemi, T. 2016. *API:t ovat modernin integraatiostrategian ydin*. Digiarjessa blogi, Digia Oyj. Luettavissa: https://blog.digia.com/rest-api. Luettu 13.5.2020.

Simonov, O. 2020. *What is Docker? Why is it important and necessary for developers? Part I*. DEV Community. Luettavissa: https://dev.to/amoniacou/what-is-docker-why-is-it-important-and-necessary-for-developers-part-i-39e5. Luettu 18.5.2020.

Tanner, G. *The definitive Guide to Docker compose*. Gabriel Tanner Blog. Luettavissa: https://gabrieltanner.org/blog/docker-compose. Luettu 18.5.2020.

Kaukoharju, A. 2019. *Gitlab-järjestelmän versionhallintatyökalut tietojenkäsittelyn opetuksessa*. Luettavissa: https://www.theseus.fi/bitstream/handle/10024/129339/Kaukoharju_Antti.pdf?sequence=1&isAllowed=y. Luettu: 21.5.2020.