Taustapalvelun ja käyttöliittymän erottaminen sekä pilvisiirtymän valmistelu

Eemeli Ristikartano

Haaga-Helia ammattikorkeakoulu
Tradenomi
Amk-opinnäytetyö
2024

Tiivistelmä

Tekijä(t)
Eemeli Ristikartano

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Taustapalvelun ja käyttöliittymän erottaminen sekä pilvisiirtymän valmistelu

Sivu- ja liitesivumäärä
36 + 0

Tämän työn aiheena on web-sovelluksen taustapalvelun ja käyttöliittymän erottaminen toisistaan tiedostotasolla. Lisäksi työssä valmistellaan toimeksiantajan web-sovelluksen infrastruktuurin siirtämistä julkiselle pilvialustalle. Tämä on laadultaan toiminnallinen opinnäytetyö. Työn tuloksena on syntynyt tuotekehitystiimille siistitty repositorio. Repositorio on säilytyspaikka projektin lähdekoodille ja muille resursseille. Lisäksi tiimi on saanut tietoa siitä, miten taustapalvelua täytyy jatkokehittää pilvisiirtymän mahdollistamiseksi.

Työn toiminnallinen osuus toteutetaan vuoden 2024 alussa ja se on kestoltaan noin kolme kuukautta. Toiminnallisen osuuden tekniset ratkaisut esitetään niiden omissa osioissaan.

Työn ensimmäinen kokonaisuus alkaa teoriaosuudella, missä esitellään ja vertaillaan eri vaihtoehtoja lähdekoodin säilyttämiselle. Teoriaosuuden perusteella on valittu sopiva repositoriomalli. Toiminnallisessa osiossa toteutetaan repositorion uudelleenjärjestely. Erottaminen suoritetaan ohjelmallisesti skriptien avulla. Tuloksena on monorepositorio, jossa taustapalvelu sekä käyttöliittymä on erotettu toisistaan omiin hakemistoihinsa.

Työn toisessa kokonaisuudessa teoriaosuudessa esitellään eri pilvipalvelumalleja. Teoriaosuuden jälkeen esitellään toimeksiantajan jatkokehityssuunnitelma pilvisiirtymän suhteen sekä perustellaan tulevan siirtymän syitä. Toiminnallisessa osuudessa on kehitetty prototyyppi sisäänkirjautumissivusta. Prototyyppi on toteutettu Reactilla ja siinä on kielenä TypeScript. Tuloksena on saatu tietoa siitä, miten taustapalvelua täytyy jatkokehittää pilvisiirtymää varten.

Viimeisessä osiossa pohditaan työn onnistumista sekä tulosten tarpeellisuutta ja saavutettuja hyötyjä. Osiossa mietitään omaa oppimista sekä arvioidaan, mitä olisi voinut tehdä paremmin. Lopuksi tuodaan esille mahdollisia jatkokehityskohtia.

Asiasanat
Lähdekoodi, repositorio, pilvipalvelut, versionhallinta

Sisällys

1 Johdanto

Kun puhutaan pitkäikäisistä projekteista, saattaa niiden lähdekoodi sisältää kymmeniä, ellei satoja tuhansia rivejä koodia. Koska koodia kertyy niin paljon, on mahdollista, että projektin rakenne muuttuu epäselväksi. Ajan lisäksi usein myös henkilöjen ja käytäntöjen vaihtuminen tuo vaikutuksia koodiin. Kooditiedostojen lisäksi projektista löytyy konfigurointiin sekä sovelluksen koontiin ja julkaisemiseen liittyviä tiedostoja. Käyttöliittymän tiedostoja saattaa löytyä projektin juuritasolta tai taustapalvelun resurssien seasta. Samalla tavalla taustapalvelun tiedostoja voi löytyä eri puolilta projektia.

Projektin siistiminen on asia, mihin ei välttämättä ole aikaa sillä se näyttäytyy aikaa vievänä ja tuottamattomana työnä. Onnistuneen siivoamisen takaamiseksi täytyy löytää ohjelmiston eri osien riippuvuudet. Tiedoston sijainnin muuttaminen tarkoittaa usein myös tiedostojen välisten polkujen päivittämistä. Näiden riippuvuussuhteiden löytäminen vaatii projektin rakenteen ja projektiin liittyvien työkalujen ja kirjastojen ymmärtämistä.

Kun lähdetään kehittämään tuotetta, olisi hyvä, että lähtötilanne olisi mahdollisimman hyvin organisoitu. Projektin rakenne vaikuttaa projektin luettavuuteen sekä eri osien rajojen hahmottamiseen. Sovelluksen koontiin tarvitaan sekä käyttöliittymän, että taustapalvelun tiedostoja. Jos sovelluksen eri osien tarvitseman resurssit ovat selkeästi erillään, on projektin jatkokehittäminen helpompaa.

Opinnäytetyön ensimmäinen osa esittelee vaihtoehdot lähdekoodin säilyttämiseen. Lähdekoodia säilytetään yleensä versionhallintatyökalun tarjoamassa repositoriossa. Repositoriorakenteessa on kaksi perinteistä lähestymistapaa. Nämä ovat monorepo ja multirepo. Monorepossa on yksi repositorio, joka sisältää kaikki moduulit, joita sovellus tarvitsee (Kaufmann 2022, luku 17). Toinen lähestymistapa on sijoittaa tuotteen eri moduulit tai palvelut omiin repositorioihinsa, jolloin puhutaan multireposta (Kaufmann 2022, luku 17). Työssä käsitellään sekä monorepon organisaatiomallia, että yhden sovelluksen monorepoa. Lisäksi työssä käsitellään kolmatta vaihtoehtoa: metarepoa. Metarepositorion avulla pystytään yhdistämään useita repositorioita yhdeksi repositorioksi (Davis 2021, luku 11.2.5). Kustakin lähestymistavasta löytyy hyviä ja huonoja puolia.

Valinta toimeksiantajan repositoriomallista tehdään opinnäytetyön ensimmäisen teoriaosuuden perusteella. Suunnitelmavaiheessa esitetään, miltä mahdollinen uuden repositorion rakenne voisi näyttää. Tämän jälkeen toiminnallisessa osiossa toteutetaan valinta sekä siistitään toimeksiantajan lähdekoodin rakenne tiedostotasolla. Projektin siistiminen tapahtuu erottamalla käyttöliittymä taustapalvelusta tiedostotasolla. Kummatkin sovelluksen osat sijoitetaan omiin hakemistoihinsa.

Siistiminen täytyy suunnitella ja toteuttaa huolellisesti, koska riskinä on ohjelmiston rikkoutuminen ja kehitystyön hidastuminen. Jos jokin riippuvuussuhde jää päivittämättä, voi se johtaa se virhetilanteeseen, jota ei välttämättä huomata heti, ja jonka korjaamiseen kuluisi aikaa. Tärkeänä tavoitteena tässä työvaiheessa on, että projektin siistiminen ei aiheuta uusia ongelmia, eikä vaikuta merkittävästi tuotteen kehittämiseen. Viimeisenä esitellään lopputulos ja pohditaan jatkokehityksen kohteita.

Opinnäytetyön toisena aiheena ovat pilvipalvelut ja pilvisiirtymän valmistelu. Organisaatioilla on useita eri vaihtoehtoja, joissa omaa palvelua voi ajaa ja ylläpitää. Organisaatiot voivat omistaa fyysisiä konesaleja, tai organisaatio voi myös ajaa sovellustaan pilvialustalla ulkoistaen samalla suuren osan ylläpitotehtävistä palveluntarjoajalle. Tunnettuja pilvipalveluja ovat esimerkiksi Google Cloud Platform, Microsoft Azure ja Amazon Web Services.

Teoriaosuudessa esitellään pilvipalveluiden historiaa, pilvipalvelumalleja sekä pilveen siirtymisen hyötyjä ja käydään läpi Googlen tarjoamia ratkaisuja. Tavoitteena on toteuttaa prototyyppi sisäänkirjautumissivusta. Suunnitelmavaiheessa kerrotaan, miten prototyyppi toteutetaan sisäänkirjautumissivusta. Tämän jälkeen toteutetaan suunnitelma ja esitellään tulokset. Prototyypin avulla saadaan tietoa siitä, miten taustapalvelua täytyy jatkokehityksen aikana muuttaa, jotta pilvisiirtymä onnistuisi. Lisäksi prototyypissä tutustutaan Googlen Firebase Hosting-palveluun.

Toimeksiantajana toimii työnantajani Zeroni Oy. Zeronin sovellus tarjoaa työkaluja ja prosessikehitystä ulkoisen työvoiman hallintaan. Toimeksiantajan lähdekoodia esittäessäni en paljasta kaikkien tiedostojen nimiä, vaan puhun yleisemmin käyttöliittymän tai taustapalvelun tiedostoista.

Tarkoituksena on, että valittua repositoriomallia ja järjesteltyä projektia käytettäisiin jatkossa tuotteen kehittämisessä. Lisäksi tavoitteena on saada tietoa siitä, miten taustapalvelua tarvitsee muuttaa pilvisiirtymään liittyen.

Opinnäytetyöstäni on hyötyä organisaatioille, joissa on samankaltaisia suunnitelmia. Uskon, että monessa vanhemmassa sovelluksessa projekti voi olla vaikeaselkoinen ja sen osia on vaikea erottaa toisistaan. Vaihtoehdot eri repositoriomalleista ovat merkityksellisiä sekä olemassa olevien, että uusien sovellusten kohdalla. Uskon myös, että monet yritykset haluavat siirtyä pilvinatiivimpaan ympäristöön, joten aihe on erittäin ajankohtainen. Pilvipalvelut mahdollistavat usean ylläpitotehtävän ulkoistamisen, jolloin tuotteen kehittämiseen jää enemmän aikaa. Toimeksiantajani saa käyttöönsä heille sopivan repositoriomallin, siistityn projektin sekä tietoa taustapalvelun muutostarpeista.

2 Lähdekoodi

Sovellusta ei ole ilman lähdekoodia. Lähdekoodin versionhallintaan on tarjolla useita eri ohjelmistoja, jotka tukevat lähdekoodin käsittelyä. Lähdekoodin rakenteen järjestyksessä pitämisessä voi olla haasteita.

Osio lähtee liikkeelle esittelemällä versionhallintaa sekä versionhallintaan tarkoitettuja työkaluja. Tämän jälkeen esitellään ja vertaillaan repositoriomalleja ja lopuksi valitaan sopiva malli toimeksiantajalle. Toiminnallisessa osiossa toteutetaan valinta sekä siistitään projektin rakenne.

2.1 Versionhallinta

Versionhallinta on tapa hallita ja pitää kirjaa lähdekoodin muutoksista. Virheen tapahtuessa lähdekoodin nykyistä tilaa voidaan verrata aiempiin, ja tämän avulla virhe voidaan paikantaa ja korjata. Versionhallinta alentaa inhimillisten virheiden riskiä. Ohjelmistokehittäjien työskennellessä tiiminä, versionhallinta mahdollistaa sen, että eri kehittäjän voivat kehittää ohjelmistoa samanaikaisesti. (Atlassian s.a. a.)

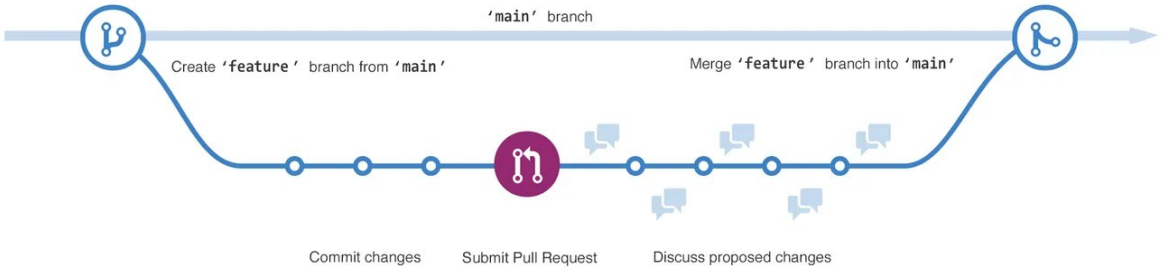
Versionhallintaohjelmisto pitää kirjaa jokaisesta muutoksesta lähdekoodissa. Versionhallintaohjelmisto tarjoaa kolme selvää etua: jokaisen tiedoston kattava historia muutoksista, haarat ja niiden yhdistäminen sekä viimeisenä jäljitettävyys. (Atlassian s.a. a.)

2.2 Versionhallinnan työkalut

Versionhallinnan tueksi on markkinoilla useampia palveluita, jotka tarjoavat käyttöliittymän versionhallinnan tueksi. Yksi tunnettu on GitHub. GitHub on yksi monista palveluista, jota voi myös käyttää komentorivityökalulla nimeltä Git (GitHub s.a. a). Tällaiset palvelut tarjoavat nykyään useita eri ominaisuuksia lähdekoodin säilömisen lisäksi. Esimerkiksi GitHub tarjoaa CI/CD (Continuous Intergration/Continuous Delivery) työkaluja, turvallisuuteen liittyviä ominaisuuksia, projektinhallintatyökaluja ja monia muita ominaisuuksia (GitHub s.a. a).

Palveluun luodaan projektille repositorio (englanniksi repository). Repositorio voi sisältää hakemistoja, tiedostoja, kuvia tai mitä vain projekti tarvitsee. (GitHub. s.a. b.). Tässä kontekstissa puhutaan projektista, joka sisältää suurimmalta osalta koodia. Haarojen (englanniksi branch) luominen mahdollistaa, että repositoriosta on useampi versio samanaikaisesti. Uusien ominaisuuksien lisääminen helpottuu haarojen avulla. Muiden haarojen muutokset eivät vaikuta toisiin haaroihin ennen kuin ne yhdistetään. Uuden haaran voi kopioida toisesta haarasta. Jos joku tekee muutoksia alkuperäiseen haaraan, kehittäjä voi yhdistää nämä muutoksen omaan haaraansa. Lopuksi oman haaran voi yhdistää päähaaraan. Se on usein nimetty main-haaraksi. (GitHub s.a. b.)

Kuvassa 1 on havainnollistettu, miten main-haarasta on luotu feature-haara. Feature-haaraa kehitetään eteenpäin erillään main-haarasta. Kuvassa 1 feature-haarasta tehdään katselmointipyyntö (englanniksi pull request), jossa muut kehittäjät voivat katselmoida muutosta ja esittää parannusehdotuksia. (GitHub. s.a. b). Lopuksi kuvassa 1 havainnollistetaan, miten feature-haara yhdistyy main-haaraan.



Kuva 1. Haarojen toimintalogiikka (GitHub s.a. b CC BY 4.0)

On mahdollista, että haarojen yhdistämisessä on konflikteja. Useimmiten konfliktit johtuvat siitä, että samaa tiedostoa on muutettu lähdehaarassa ja muutokset osuvat tulohaaran kanssa samoille riveille. GitHub kertoo näistä konflikteista ja estää haarojen yhdistämisen, ennen kuin konfliktit ovat korjattu (GitHub s.a. b). Tässä nousee hyvin esille, miten versionhallintaohjelmistot varmistavat muutosten eheyden ja vähentävät inhimillisten virheiden määrää.

Versionhallintaohjelmisto on tärkeä ja jopa välttämätön työkalu ohjelmiston lähdekoodin ylläpidossa ja sen käyttäminen tuo varmuutta siihen, että lähdekoodi pysyy tallessa. Repositorio voi sisältää yhden tai useamman projektin. Tämän valinnan tekeminen voi vaikuttaa lähdekoodin ylläpidettävyyteen.

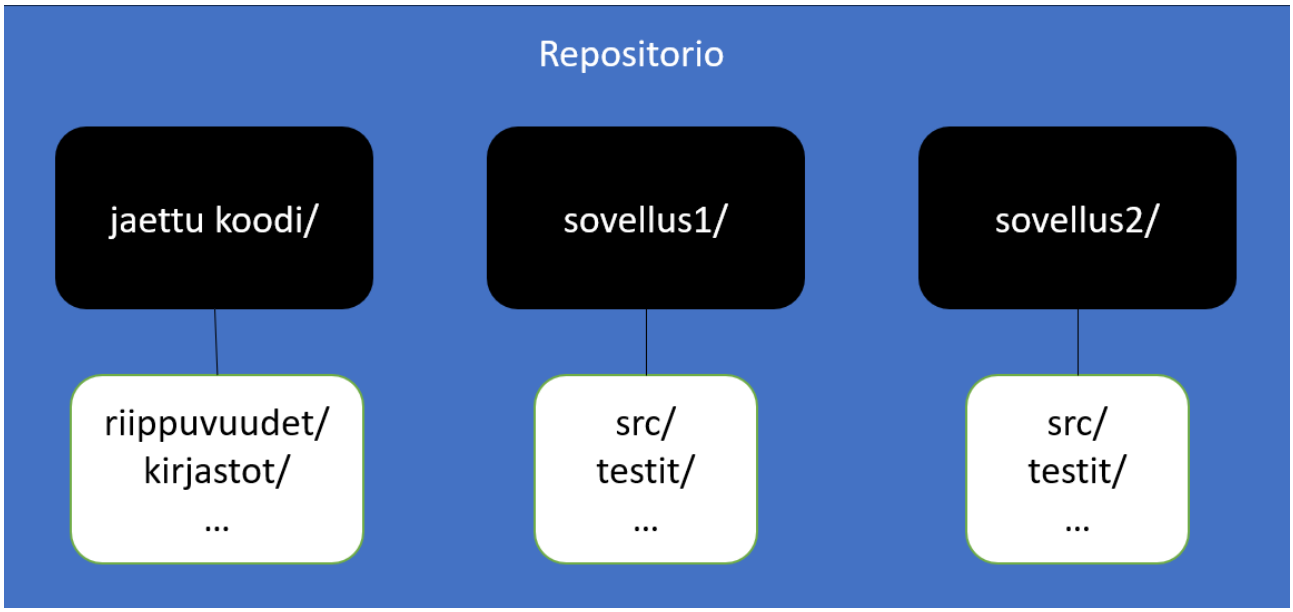
2.3 Monorepo

Monorepoa voi lähestyä kahdella eri tavalla. Ensimmäisessä lähestymistavassa organisaation kaikkien sovellusten lähdekoodit ja organisaation omat kirjastot löytyvät samasta repositoriosta. Toisessa lähestymistavassa jokaisella sovelluksella on oma repositorionsa, kuitenkin niin, että jokainen näistä repositorioista sisältää siihen sovellukseen kuuluvan lähdekoodin ja riippuvuuden. Kappaleen teoriaosuudessa käsitellään monorepon hyötyjä organisaatiomallin kautta.

Organisaatioilla on usein käytössään monorepo eli monerepositorio. Tämä repositorio sisältää kaiken lähdekoodin. Sovellukset, komponentit ja kirjastot, joita käytetään yleisesti kehittämisessä. Lähdekoodin pitäminen yhdessä ja samassa repositoriossa tuo useita etuja. Kaikki riippuvuudet

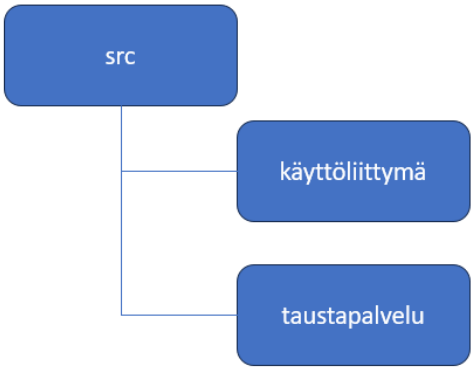
pysyvät samassa tahdissa, eikä niitä tarvitse päivitellä erikseen eri projekteihin. (Roldàn 2023, luku 14.)

Kuvassa 2 esitetään, miltä organisaation monorepo voisi näyttää. Repositorio sisältää kaksi eri sovellusta ja niiden lähdekoodit. Lisäksi repositorio sisältää jaetun koodin. Jaetun koodin hakemisto sisältää erilaisia riippuvuuksia ja kirjastoja, joita koko organisaatio käyttää.



Kuva 2. Monorepon organisaatiomalli

Kuvassa 3 kuvataan, miltä monorepon rakenne voi näyttää yhden sovelluksen osalta. Src-hakemisto sisältää käyttöliittymän sekä taustapalvelun.



Kuva 3. Yksinkertainen yhden sovelluksen monorepo

Monorepo sisältää useita muitakin etuja. Monorepon avulla lähdekoodin jakaminen on helppoa. Lähdekoodia pystyy hyödyntämään useissa eri projekteissa. Monorepo auttaa siinä, että jokainen

monorepon projekti käyttää samoja jaettuja komponentteja. Tämä vähentää riskiä versioiden yhteensopimattomuudesta. (Roldàn 2023, luku 14.)

Tilanteita, joissa voisi tulla ongelmia eri versioiden kanssa on monia. Voi olla, että käyttöliittymään on kehitetty ominaisuus, jota ei ole vielä kehitetty taustapalveluun. Tällaisen ominaisuuden käyttäminen käyttöliittymässä voi aiheuttaa virhetilanteen. Monorepon ja haarojen avulla voidaan olla varmempia siitä, että käyttöliittymäversio on yhteensopiva taustapalvelun kanssa. Sama pätee toiseen suuntaan.

Muutosten tekeminen monorepossa on suoraviivaisempaa. Monorepon sisällä voi tehdä koko projektiin vaikuttavia muutoksia ilman, että niitä täytyy tehdä niitä yksittäisiin repositorioihin. Lisäksi koko tiimillä on pääsy samaan keskitettyyn repositorioon. Monorepon avulla projektin päivittäminen on nopeampaa. Monorepo nopeuttaa tuotekehitystä, koska kehittäjä pystyy päivittämään kaikkia projekteja yhdessä repositoriossa. Tiimin jäsenet näkevät suoraan, mitä muutoksia toinen on tehnyt projekteihin. (Roldàn 2023, luku 14.)

Useat isot ohjelmistoyritykset ovat siirtyneet organisaatiomalliin. Näistä yksi esimerkki on Google. (Jaspan ym. 2018, 225) Google käyttää räätälöityä monoliittista repositorioa. Tätä repositorioa käyttää suurin osa yrityksen sovelluskehittäjistä. Artikkelin tekoaikaan repositorio sisälsi arviolta miljardi tiedostoa. (Potvin & Levenberg 2016, 78.)

Jaspanin ja kumppanien (2018, 232) tutkimuksessa haastatteluissa nostettiin hyötyjä ja haittoja monoreposta. Monorepon tuoma näkyvyys on tehokasta. Mahdollisuus etsiä koodia vaikuttaa positiivisesti koodin laatuun sekä koodaamisen nopeuteen. Kehittäjät pystyvät etsimään koodia koko organisaation lähdekoodista. Monorepo mahdollistaa, että koodilla on pysyvä tyyli, joka yhdistettiin laadukkaaseen koodiin. Hyödyksi nostettiin myös riippuvuuksien hallinta. (Jaspan ym. 2018, 232–233.)

Vaikka kappaleessa käsiteltiin monorepoa nimenomaan organisaatiomallin kautta, nämä hyödyt ovat olemassa myös yksittäisten sovellusten monorepo mallissa. Yksittäisten sovellusten monorepon avulla saa suuren osan organisaatiomallin hyödyistä, ilman että repositorion koko kasvaa liian suureksi.

2.4 Multirepo

Toinen lähestymistapa koodin säilyttämiseen on multirepositorio eli multirepo. Tässä mallissa jokainen moduuli tai mikropalvelu on omassa repositoriossaan. Toimivan sovelluksen julkaisemiseksi täytyy siis ottaa huomioon useampi koodin repositorio. (Kaufmann 2022, luku 17.) Lisäksi täytyy varmistua repositorioissa olevien versioiden yhteensopivuudesta.

Suurin etu multirepossa on se, että se vähentää monimutkaisuutta yksittäisissä koodin repositorioissa. Jokaista repositoriota voidaan hallita autonomisesti. Lisäksi yksittäinen koodin repositorio voidaan koota ja julkaista erillään muista. (Kaufmann 2022, luku 17.)

Kuvassa 4 havainnollistetaan yksinkertaista multirepo-mallia. Kuvassa 4 on sovellus, joka tarvitsee käyttöliittymän ja taustapalvelun toimiakseen. Kumpikin palvelu on omassa repositoriossaan.

Kuva 4. Yksinkertainen multirepo

2.5 Metarepo

Metarepon eli metarepositorion avulla pystytään yhdistämään yksittäiset repositoriot yhdeksi repositorioksi. Metarepoa voi ajatella ikään kuin virtuaalisena repositoriona. Metarepon avulla pystytään säilyttämään monorepon tuoma yksinkertaisuus sekä mukavuus. Sen avulla ei kuitenkaan tarvitse luopua multirepon tuomasta joustavuudesta. (Davis 2021, luku 11.2.5.)

Git-työkalun avulla voi luoda metarepon, joka sisältää alimoduuleina kaikki muut repositoriot (Kaufmann 2022, luku 17). Vaihtoehtona on myös asentaa erillinen työkalu nimeltä meta. Se on saatavilla osoitteessa https://github.com/mateodelnorte/meta. Meta-työkalulla metarepo tehdään luomalla .meta-konfiguraatiotiedosto, joka sisältää listan erillisistä repositorioista (Davis 2021. luku 11.2.5). Kuvassa 5 on esimerkki siitä, miltä .meta-konfiguraatiotiedosto näyttää.

```
{
  "projects": {
    “exampleService1": "git@github.com:<organization>/exampleService1.git",
    " exampleService2": "git@github.com:< organization >/exampleService2.git",
    " exampleService3": "git@github.com:< organization >/exampleService3.git"
  }
}
```

Kuva 5. Esimerkki .meta-konfiguraatiotiedostosta

2.6 Repositoriomallien vertailu

Repositorion strategian valintaa on hyvä miettiä sekä tiimien, että yksittäisen ohjelmoijan näkökulmasta. Jossakin organisaatiossa voi olla useita tiimejä, kun taas toisessa yksi tai kaksi tiimiä. Tässä vertailussa monorepoa käsitellään näkökulmasta, jossa monoreposta löytyy yhden sovelluksen lähdekoodi. Kuvat 3 ja 4 esittävät, miltä tämän vertailun kuvitteellinen kehitettävä sovellus voisi näyttää.

Monorepo nopeuttaa ja helpottaa yhteistyötä sekä koodin jakamista. Ohjelmoijan ei tarvitse hakea useaa eri repositoriota itselleen, eikä hänen tarvitse yhdistää muutoksia eri repositorioihin, koska kaikki projektin koodin on samassa repositoriossa. Varsinkin pienemmissä projekteissa monorepo on hyvä vaihtoehto. (Scholl, Jausovec & Swanson 2019, luku 5.)

Monorepo jopa pakottaa tiimien väliseen yhteistyöhön. Tiimien täytyy sopia, miten koodia ylläpidetään. Lisäksi jokainen näkee muiden koodin. Tämä vähentää riskiä siinä, että tiimien jäsenet eristäytyisivät liikaa. Monorepo voi auttaa toistamaan hyviä pohjia koodille, joita muut tiimit ovat tuottaneet. (Vinci 2023, luku 3.)

Toisaalta ohjelmoija ei välttämättä kuitenkaan ole kehittämässä jokaista osiota sovelluksessa. Siksi on myös hyvä miettiä, onko tarpeellista, että ohjelmoija hakee koko sovelluksen lähdekoodin itselleen. Toinen vaihtoehto on se, että ohjelmoija hakee itselleen vain tarpeellisen repositorion. Multirepo mahdollistaa tämän. Lisäksi multirepo helpottaa siinä, että projektin koodiin ei tule liikaa sisäisiä riippuvuuksia. Monorepo voi ajaa siihen tilanteeseen, että koodia uudelleen käytetään liikaa. Tämä lisää koodin välisiä riippuvuuksia. Liiallinen uudelleen käyttäminen voi johtaa vaikeasti hallittaviin riippuvuuksiin. (Scholl, Jausovec & Swanson 2019, luku 5.) Yleisesti ottaen koodin uudelleenkäyttö on hyvä asia. Vaarana on se, että uudelleenkäytettävää koodia muutetaan ottamatta huomioon kaikkia osia, jotka koodia käyttävät. Varomaton muutos saattaa aiheuttaa virhetilanteen muualla.

Lisäksi multirepoilla tiimit voivat tiimikohtaiset säännöt ja ohjeet sille, miten koodia ylläpidetään. Toisaalta tämä lähestymistapa voi lisätä tiimien välistä eriytymistä, koska jokainen tiimi on keskittynyt omaan repositorioonsa (Vinci 2023, luku 3.)

Monorepon avulla kaikkia sovelluksen koontiin liittyviä työkaluja voidaan ylläpitää keskitetysti. Tämä voi vähentää toisteista työtä. Multirepoissa taas jokaiseen repositorioon täytyy luoda omat työkalut. Kokeneilla tiimeillä löytyisi optimoidut työkalut, kun taas vähemmän kokeneilla tiimeillä voisi olla haasteita näiden työkalujen luomisessa. (Vinci 2023, luku 3.) Jos tavoitteena on päästä kohti pilvinatiivia ympäristöä, olisi hyvä pyrkiä tilanteeseen, jossa jokaisen palvelun voi julkaista erikseen (Scholl, Jausovec & Swanson 2019, luku 5). Toisaalta myös monorepon voi toteuttaa tavalla, jossa eri palvelut voi julkaista erikseen.

Eli onko ratkaisu kaikkiin huonoihin puoliin metarepo? Ei välttämättä. Uuden työkalun tai kolmannen osapuolen kirjaston käyttöönotto vaatii tiimiltä aikaa perehtymiseen. Yksi vaihtoehto on ottaa käyttöön Git-työkalun alimoduulit. Alimoduulit mahdollistavat repositorion pitämisen toisen repositorion alihakemistona. Tämän avulla on mahdollista kloonata toinen repositorio omaan projektiin. (Chacon & Straub 2014, luku 7.11.)

Vaihtoehtona on myös käyttää kolmannen osapuolen tarjoamaa työkalua. Suosittujen avoimen lähdekoodin sovelluksissa on paljon hyviä puolia. Niiden käyttö on usein helppoa, ne saattavat nopeuttaa oman sovelluksen kehittämistä, ne eivät maksa mitään ja ne ovat laadukkaita (snyk s.a.).

Avoimen lähdekoodin sovellukset eivät ole kuitenkaan ongelmattomia. Nekin sisältävät tietoturvariskejä. Usein avoimen lähdekoodin sovelluksella on myös omia riippuvuuksiaan. Sen lisäksi, että pitäisi seurata omien riippuvuuksien haavoittuvuuksia, pitäisi seurata avoimen lähdekoodin riippuvuuksien haavoittuvuuksia. Toinen ongelma on lisenssit. Lisenssiehdot voivat olla vaikeaselkoisia, ja ne voivat muuttua, joten tilannetta täytyy myös seurata. Viimeisenä on hyvä muistaa, että avoimen lähdekoodin pakettia saattaa ylläpitää vain yksi ohjelmoija tai pieni tiimi. Voi myös olla, että ylläpito on jo päättynyt. (snyk s.a.)

Pakettien ja riippuvuuksien valinta vaatii aikaa tiimiltä. Täytyy varmistua, että paketti on turvallinen ja, että sitä ylläpidetään. Lisäksi paketteja ja riippuvuuksia täytyy päivittää, jotta ne eivät muodostu tietoturvariskeiksi. Omista riippuvuuksista ja kirjastoista kannattaa seurata ainakin kahta mittaria: kuinka monta päivää on haavoittuvuuden löytämisen ja sen korjaamisen välillä ja kuinka kauan keskimääräisesti kuluu aikaa ilmoitetun ongelman korjaamisessa (snyk s.a.).

Otetaan esimerkiksi jo aiemmin mainittu Meta-työkalu. Metan GitHub sivulta selviää, että sen viimeisin versio on julkaistu 28.4.2021. Lisäksi sivulta löytyy 17 ilmoitettua ongelmaa, joista vanhimmat ovat vuodelta 2021. Myös avoimia pull requesteja on 8. Paketin package.json-tiedostosta voi

löytää paketin riippuvuuksia dependencies-osiosta. (GitHub s.a. c.) Kuvassa 6 on osa paketin riippuvuuksista.

Kuva 6. Meta-paketin package.json tiedostosta kuva riippuvuuksista

Kuvasta 6 löytyvä tabtabin viimeisin version on vuodelta 2016. Sen GitHub sivulla on 21 avointa ongelmaa sekä saman verran avoimia pull requesteja. (GitHub s.a. d.) En väitä, että Meta-paketti olisi vaarallinen. Jokaista riippuvuutta täytyy tarkastella tilannekohtaisesti ja itse päättää, onko riippuvuuden vanhentuminen kriittinen asia. Pakettien ja riippuvuuksien tarkastaminen voi olla työlästä. Toki tähänkin löytyy työkaluja, joilla tarkistuksia voi automatisoida.

Lopuksi on hyvä mainita, että turvallisuusriskien lisäksi jokainen kirjasto, paketti tai riippuvuus on yksi elementti lisää, joka voi hajota ja näin ollen hidastaa kehitystyötä. Paketit lisäävät myös sovelluksen kokoa, mikä on hyvä ottaa huomioon.

Jokaisesta lähestymistavasta löytyy hyvää ja huonoa. Monorepo voi kasvaa suureksi ja siitä voi tulla vaikeasti hallittava. Multirepossa taas poistuu mahdollisuus hakea koodia koko sovelluksesta ja tiimien tekeminen voi siiloutua. Metarepo hieman eksoottisempana vaihtoehtona vaatii uuteen asiaan perehtymistä ja mahdollisesti kolmannen osapuolen kirjaston, joka voi taas synnyttää uusia ongelmia.

3 Zeronin lähdekoodin rakenne

Zeroni-sovelluksen lähdekoodi on monorepossa. Tutkimuksessa ei löytynyt perusteita lähteä muuttamaan repositorion rakennetta. Sovellusta on kehittämässä kuusi tiimin jäsentä. Tiimistä osa keskittyy käyttöliittymän kehittämiseen, osa taustapalvelujen kehittämiseen ja osa infran ylläpitämiseen. Osa tekee sekä käyttöliittymän, että taustapalvelun kehittämistä. Sovelluksessa käytetään Apache Maven -työkalua. Päätös monorepossa pysymisestä tarkoittaa sitä, että toiminnallisessa osiossa keskitytään projektin rakenteen siivoamiseen.

Valinta monorepossa pysymiseen myötäilee teoriaosuudessa mainittuja etuja, joita monorepo tuo. Tämän valinnan ansiosta kaikilla on pääsy samaan lähdekoodiin. Jokainen pystyy näkemään, mitä muutoksia sovelluksen eri osioihin on tullut tai on tulossa. Monorepo helpottaa sovelluksen koontia ja julkaisemista, koska kaikki tarvittavat resurssit löytyvät samasta sijainnista. Monorepon avulla kehittäjien ei tarvitse tuoda omaan kehitysympäristöön useita eri repositorioita. Tiimin sisälle ei haluta muodostuvan siiloutumista, joka on vaarana multirepossa. Metarepoa ei valittu, koska sen tarjoamat hyödyt eivät olisi tuoneet lisäarvoa kehitystyöhön.

3.1 Vastuualueiden sekoittuminen

Zeroni-sovelluksen lähdekoodilla on pitkä historia. Tämä tarkoittaa sitä, että koodia on ehtinyt kertymään projektiin runsaasti. Projektissa käyttöliittymän ja taustapalvelun koodi on päässyt tiedostotasolla sekoittumaan.

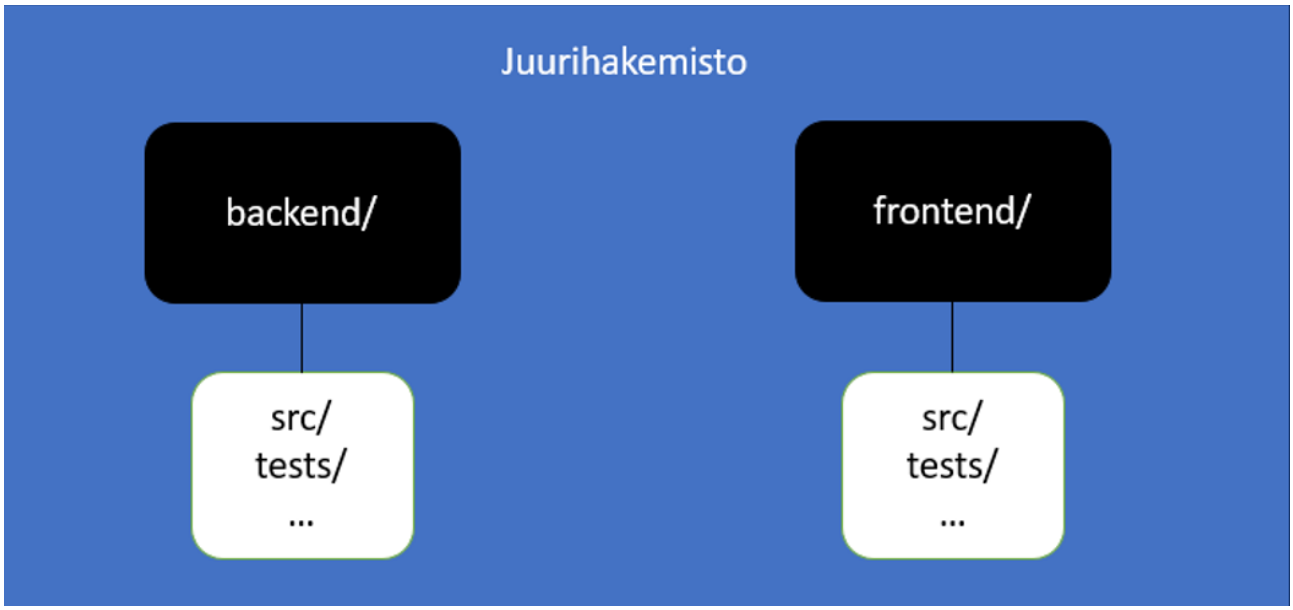
Projektin juuritasolla on paljon hakemistoja ja erilaisia konfiguraatiotiedostoja, joiden merkitys jää monille kehittäjille arvoitukseksi. Ongelmana on siis projektin rakenteen epäselvyys. Projekti halutaan siistiä ja siitä halutaan tehdä helppolukuisempi.

3.2 Suunnitelma lähdekoodin uudelleen järjestämiseksi

Käyttöliittymän ja taustapalvelun erottamiseksi lähdekoodin tasolla on hyvä tehdä suunnitelma ja miettiä tapaa, miten erottamisen toteuttaa. Suunnitelmaan liittyy neljä keskeistä vaihetta.

Ensimmäisessä vaiheessa täytyy päättää, mikä tulisi olemaan haluttu lopputulos. Juuritasolle olisi hyvä sijoittaa vain sinne tarvittavat asiat. Nämä ovat tiedostoja, jotka ovat koko projektille yhteisiä. Käyttöliittymän hakemistoon tulisi sijoittaa vain käyttöliittymään kuuluvat tiedostot. Näitä tiedostoja ovat esimerkiksi käyttöliittymän lähdekoodi, käyttöliittymässä tarvittavat kirjastot ja muut riippuvuudet sekä käyttöliittymän koontiin liittyvät tiedostot. Taustapalvelujen hakemistoon kuuluu luonnollisesti taustapalveluihin liittyvät kirjastot, riippuvuudet ja taustapalvelun lähdekoodi sekä taustapalvelun koontiin liittyvät tiedostot.

Kuvassa 7 kuvataan mahdollista lopputulosta. Tavoitteena on, että uudessa mallissa juurihakemistosta löytyisi kaksi alihakemistoa. Nämä olisivat backend (taustapalvelu) sekä frontend (käyttöliittymä). Backend-hakemistosta löytyisi taustapalveluun liittyvät asiat. Kuvassa esimerkkeinä ovat src- sekä tests-hakemistot. Toinen juuritasolta löytyvä alihakemisto on frontend eli käyttöliittymän hakemisto. Sinne sijoitetaan käyttöliittymään liittyvän asiat. Esimerkkeinä ovat src- ja tests-hakemistot. Kuvaan ei ole laitettu kaikkia juurihakemistoon jääviä tiedostoja. Todennäköisesti sinne tulee jäämään koko sovellukseen liittyviä tiedostoja.



Kuva 7. Projektin rakenteen mahdollinen lopputulos

On hyvä huomioida, että joidenkin tiedostojen ja hakemistojen välillä voi olla haastavaa päättää mihin hakemistoon ne kuuluvat. Muutokset rakenteessa eivät ole kuitenkaan lopullisia ja niitä voidaan muuttaa tarpeen vaatiessa.

Toinen suunnittelun paikka on keksiä keino, miten lähdekoodista voidaan löytää käsin muutettavat arvot. Esimerkiksi sovelluksen koontiin liittyvistä tiedostoista, konfigurointiin liittyvistä tiedostoista sekä muista vastaavista tiedostoista löytyy polkuja, jotka täytyy päivittää käsin. Ohjelmointiympäristöt huolehtivat automaattisesti osasta polkujen päivittämisestä. Kaikki polut eivät kuitenkaan päivity itsestään. Joissain tilanteissa polku on annettu merkkijonona ja kirjoitettu itse tiedostoon. Tällaisia tilanteita ohjelmointisovellus ei tunnista, eikä täten osaa muuttaa.

Kolmantena täytyy varmistua siitä, että halutut toiminnot toimivat. Sovellus täytyy pystyä kääntämään, sitä täytyy pystyä kehittämään ja se täytyy saada käyntiin paikallisesti. Se pitää pystyä julkaisemaan samaan tapaan kuin aiemminkin. Yleisimmät komennot ja toiminnot tullaan testaamaan läpi.

Viimeisenä täytyy suunnitella, miten uudistettu rakenne tuodaan päähaaraan versionhallinnassa, pysäyttämättä kehittämistä pitkäksi aikaa. Uutta rakennetta ei voi yhdistää päähaaraan vapaasti, koska kehittäjillä saattaa olla kesken asioita, jotka on tehty vanhan rakenteen mukaan.

Projektiin tullaan tekemään hakemisto, joka sisältää kaikki tarvittavat tiedostot projektin uudelleenjärjestelemistä varten. Avoimet katselmointipyynnöt käydään läpi ja niiden haarat viedään päähaaraan. Lopuksi edellä mainitun hakemiston sisältävä haara yhdistetään päähaaraan ja sen avulla projekti järjestetään uudelleen. Tämä vaihe suoritetaan tuotantoon vientiä seuraavana päivänä.

3.3 Lähdekoodin järjesteleminen

Toteutusvaihe aloitettiin tutustumalla projektin nykyiseen rakenteeseen. Projektista löytyi Dockerfile-tiedosto. Dockerfile-tiedosto sisälsi sovelluksen koontiin liittyviä vaiheita. Sen avulla oli helpompi lähteä liikkeelle, koska siitä näki karkealla tasolla, mitä tiedostoja kuuluu frontend-hakemistoon ja mitä backend-hakemistoon. Silmiinpistävää tiedostossa oli pitkä luettelo käyttöliittymän koontiin vaadituista tiedostoista. Tiedostoja on jouduttu listaamaan yksittäisinä projektin eri sijanneista. Tämä on yksi konkreettinen asia, mitä uudelleenjärjestely ratkaisee.

Projektin järjestelemiseen oli kaksi hyvää vaihtoehtoa. Ensimmäinen vaihtoehto olisi järjestellä projekti uudelleen IDE:n avulla. IDE (Integrated Development Environment) eli ohjelmointiympäristö tarjoaa hyvän tuen tiedostojen ja hakemistojen siirtelyyn. Siirrettäessä tiedoston tai hakemiston, IDE pystyy usein päivittämään tiedoston riippuvuuden polut automaattisesti. Tämä lähestymistapa vaatisi kuitenkin sen, että muutokset olisi tehtävä useaan otteeseen käsin. Siirtelyä joutuisi todennäköisesti kokeilemaan useaan otteeseen ennen viimeistä suoritusta. Lopulta päädyttiin tekemään skripti uudelleen järjestämistä varten.

Skriptin rakentaminen vaati usean kokeilun ennen kuin se toimi. Kutenkin sen avulla sai luottavaisen olon siitä, että tiedostot siirtyvät oikein ja polut päivittyvät oikein. Lisäksi skriptin avulla järjestelemin tapahtui sekunneissa. Skriptin avulla inhimillisten virheiden määrä väheni, koska skripti suorittaa aina samat operaatiot samassa järjestyksessä.

Skriptin toimivuutta joutui testaamaan useita kymmeniä kertoja. Skriptin suorittamisen jälkeen kokeiltiin kääntää ja käynnistää sovellus. Koska asiat tapahtuvat aina samassa järjestyksessä, virheen korjaaminen on helpompaa. Jos tiedostoja siirtäisi käsin, olisi mahdollista, että jokin tiedosto jäisi välistä. Tämä muuttaisi jokaista iteraatiota kokeiluvaiheessa, jolloin emme voisi olla täysin varmoja siitä, mistä virhe johtuu.

Virheen sattuessa piti osata tulkita virheviestiä tai muulla tavoin etsiä syy virheelle. Virheviestiä ei välttämättä aina tullut. Useissa tapauksissa sovellus kääntyi ja käynnistyi ongelmitta, mutta

sovelluksen käyttöliittymä ei silti toiminut. Näiden ongelmien ratkaiseminen vaati Apache Maven-työkalun dokumentaation lukemista sekä projektin rakenteeseen syventymistä.

Aluksi oli vain yksi skripti, joka aluksi siirsi tiedostot oikeisiin kohteisiin ja lopuksi päivitti riippuvuuksien polut tiedostoista. Tämä skripti päätettiin jakaa kahteen osaan. Erottamisen avulla versionhallintaan saatiin selkeämpi jälki muutoksista. Ensimmäinen muutos oli tiedostojen siirtäminen ilman muokkaamista ja toinen muutos oli polkujen päivittäminen. Kuvassa 8 on skripti, jolla siirrettiin tiedostot oikeisiin hakemistoihin. Kuvassa tiedostojen nimiä on muutettu ja tiedostojen listausta on lyhennetty.

Kuvan 9 skripti päivitti käyttöliittymän tiedostojen riippuvuuksien polut oikeiksi. Kuvan joidenkin hakemistojen nimiä on hieman muokattu. Järjestyksenä oli ajaa aluksi kuvan 8 skripti ja tämän jälkeen kuvan 9 skripti. Taustapalvelun riippuvuuksien polkuja ei tarvinnut päivittää.

```bash
#!/usr/bin/env bash
set -ev

FRONTEND_ROOT="frontend"
BACKEND_ROOT="backend"
REORG_RESOURCES="repository_reorganization_202402"

# Create target directories
mkdir -p $FRONTEND_ROOT/tests
mkdir -p $BACKEND_ROOT

# Copy new resources
cp $REORG_RESOURCES/newResource1 .
cp $REORG_RESOURCES/newResource2 .
cp $REORG_RESOURCES/newResource3 .
cp .gitignore $FRONTEND_ROOT/
cp .gitignore $BACKEND_ROOT/

# Frontend
files_to_move_frontend=(
  "frontendDirectory1"
  "frontendFile1"
  "frontendFile2"
  "..."
)

# Move frontend files and directories
for item in "${files_to_move_frontend[@]}"; do
  mv "$item" "$FRONTEND_ROOT"
done

# Additional moves
mv ".configFile1" "$FRONTEND_ROOT/.configFile1"
mv "tests/testDirectory1" "$FRONTEND_ROOT/tests"
mv "tests/testDirectory2" "$FRONTEND_ROOT/tests"
mv "tests/testFile1" "$FRONTEND_ROOT/tests"
mv "tests/.configFile2" "$FRONTEND_ROOT/tests"

echo "Files and directories moved successfully to $FRONTEND_ROOT"

# Backend
files_to_move_backend=(
  "backendDirectory1"
  "backendFile1"
  "backendFile2"
  "..."
)

# Move backend files and directories
for item in "${files_to_move_backend[@]}"; do
  mv "$item" "$BACKEND_ROOT"
done

echo "Files and directories moved successfully to $BACKEND_ROOT"
```

Kuva 8. 1_move_files.sh tiedosto

Taulukossa 1 on esitetty komennot ja niiden argumentit, joita kuvan 8 skriptissä on käytetty. Komennoilla on useita vaihtoehtoisia argumentteja, joita voi antaa.

Taulukko 1. Kuvan 8 komennot

Komento	Selite
set -ev	Set komennolla voi asettaa tai purkaa optioita ja sijaintiparametreja. Optio e määrittää sen, että virhetilanteessa skriptin suorittaminen pysähtyy. V optio tulostaa annetut komennot. (Ubuntu Manpage s.a. a.)
mkdir -p	Mkdir komento luo hakemiston. Argumentti p luo emohakemistot, jos niitä ei ole. Lisäksi argumentti mahdollistaa sen, että ei tule virhettä, jos luotava hakemisto on jo olemassa. Argumenttien jälkeen annetaan luotavan hakemiston nimi (Ubuntu Manpage s.a. b.)
cp	Cp komennolla voi kopioida tiedostoja ja hakemistoja. Mahdollisten argumenttien jälkeen tulee tiedosto tai hakemisto, joka halutaan kopioida. Viimeiseksi tulee kohde, johon resurssit halutaan kopioida. (Ubuntu Manpage s.a. c.)
mv	Mv komento siirtää tai uudelleen nimeää tiedostot. Mahdollisten argumenttien jälkeen annetaan tiedostot, jotka halutaan siirtää. Viimeiseksi tulee kohde, johon resurssit halutaan siirtää. (Ubuntu Manpage s.a. d.)
echo	Echo näyttää tekstiä tulosteessa. Mahdollisten argumenttien jälkeen annetaan merkkijono. (Ubuntu Manpage s.a. e.)

```
#!/usr/bin/env bash
set -ev

FRONTEND_ROOT="frontend"
BACKEND_ROOT="backend"
REORG_RESOURCES="repository_reorganization_202402"

# Update paths in JavaScript and TypeScript files
pushd $FRONTEND_ROOT
find ./webapp/directory1 -type f \( -name 'index.js' \) -exec sed -i 's|../../../../node_modules|../../node_modules|g' {} \;
find ./webapp/directory2 -type f \( -name 'index.js' \) -exec sed -i 's|../../../../scss|../../scss|g' {} \;
find . -maxdepth 1 -type f \( -name 'package.json' \) -exec sed -i 's|src/main/webapp|webapp|g' {} \;
find . -maxdepth 1 -type f \( -name 'tsconfig.json' \) -exec sed -i 's|src/main/webapp|webapp|g' {} \;
find . -maxdepth 1 -type f \( -name 'webpack.common.js' \) -exec sed -i 's|src/main/webapp/|webapp/|g' {} \;
find . -maxdepth 1 -type f \( -name 'webpack.common.js' \) -exec sed -i 's|src/main/webapp|webapp|g' {} \;
find ./tests -maxdepth 1 -type f \( -name 'lazyload.js' \) -exec sed -i 's|/src/main/webapp|/webapp|g' {} \;
find ./tests/unit -type f \( -name '*.js' \) -exec sed -i 's|/src/main/webapp|/webapp|g' {} \;
find ./utilities -type f \( -name '*.js' \) -exec sed -i 's|/src/main/webapp|/webapp|g' {} \;
find ./utilities -type f \( -name '*.js' \) -exec sed -i 's|src/main/webapp|webapp|g' {} \;
find ./jest -type f \( -name 'import-lib.js' -o -name 'testAppIndex.js' -o -name 'jest.config.common.js' \) -exec sed -i 's|/src/main/webapp|/webapp|g' {} \;
find ./scss -type f \( -name '_common.scss' -o -name '_lib.scss' -o -name '_top-nav.scss' \) -exec sed -i 's|/src/main/webapp|/webapp|g' {} \;
find ./.storybook -type f \( -name 'main.js' \) -exec sed -i 's|src/main/webapp|webapp|g' {} \;
popd

echo "Frontend paths updated successfully."

```

Kuva 9. 2_modify_files.sh tiedosto

Kuvassa 8 ja 9 on useita eri komentoja, joilla siirretään tiedostoja tai määritellään polkuja uudelleen. Kuvassa 9 olevat tiedostot ovat käyttöliittymän lähdekoodia sisältäviä tiedostoja. Taulukossa 2 on lueteltu komennot ja kerrottu, mitä ne tekevät.

Taulukko 2. Kuvan 9 komennot

Komento	**Selite**
pushd	Pushd komennolla voi tallentaa nykyisen hakemiston pinoon ja siirtyä uuteen hakemistoon (Kumari 2015, luku 3).
popd	Popd komennolla voi palata edelliseen hakemistoon, joka on pinon päällimmäisenä (Kumari 2015, luku 3).
find	Find komennolla voi hakea tiedostoja hakemistoista (Ubuntu Manpage s.a. f). Kuvassa 9 aluksi määritellään hakemisto, jossa haku suoritetaan. Alla on lueteltuna käytetyt argumentit ja niiden selitteet Ubuntu Manpage:n (s.a. f.) mukaan. – Maxdepth määrittelee, miltä tasolta tiedostoja etsitään. – Type määrittelee tiedostotyypin. F tarkoittaa tavallista tiedostoa. – Name komento määrittelee tässä tapauksessa haetun tiedoston nimen. – O komento on tai-ehto. – Exec komento suorittaa tässä sed komennon.
sed	Sed on editori, jolla voi filtteröidä ja muokata tekstiä. Pelkkä i komento muokkaa tiedostoa ilman, että siitä luodaan toista tiedostoa tai varmuuskopiota. (Ubuntu Manpage s.a. g.) Viimeisenä määritellään mitä muokataan ja miten. Kaikissa tapauksissa relatiivisia polkuja muokataan, jotta ne toimivat uuden hakemistorakenteen kanssa.

Skriptien tekemisen jälkeen ongelmaksi muodostui sovelluksen paikallinen kehittäminen. Sovelluksen käyttöliittymä ei toiminut, kun sovelluksen käynnisti IDE:n kautta. Syynä oli Maven-projektin rakenne. Maven projektissa verkkosovellukseen tarvittavat resurssit tulee löytyä hakemistosta src/main/webapp (The Apache Software Foundation 2023). Toinen ongelma oli sovelluksen kääntämisessä. Sovelluksen käännöksessä Maven ajoi käyttöliittymälle olennaiset komennot. Nämä komennot olivat määritelty pom.xml-tiedostossa ja komennot asensivat käyttöliittymän riippuvuuden ja suoritti käyttöliittymän koonnin. Vaikka pom.xml-tiedostoon päivitti käyttöliittymän hakemistoksi frontend-hakemiston, kääntäminen ei onnistunut halutulla tavalla, koska koko src/main/webapp hakemistoa ei enää ollut uudessa rakenteessa.

Ratkaisu oli luoda symbolinen linkki. Paikallista koontia varten luotu skripti ajaa käyttöliittymälle tarpeelliset komennot, luo symbolisen linkin ja lopuksi ajaa mvn compile komennon. Symbolisen linkin voi myös luoda manuaalisesti, kuvan 10 rivin 11 mukaisesti. Kuvassa 10 on tiedosto, joka luo symbolisen linkin ja kääntää sovelluksen. Ratkaisun avulla saatiin poistettua pom.xml-tiedostosta käyttöliittymän koontiin liittyvä komennot, mikä edistää tavoitetta käyttöliittymän ja taustapalvelun

erottamisessa. Lisäksi sovelluksen julkaisemiseen liittyvissä tiedostoissa päivitettiin polkuja, jotta ne toimisivat jatkossakin.

```
#!/usr/bin/env bash
set -ev

# build frontend
pushd frontend
npm install
npm run build
popd

# link frontend output to backend
ln -sfT ../../../frontend/webapp backend/src/main/webapp

# build backend
pushd backend
if [ "$1" == "clean" ]; then
  mvn clean compile
else
  mvn compile
fi
popd
```

Kuva 10. local-compile.sh tiedosto

Tiimin kanssa sovittiin päivä, jolloin avoimet kehityshaarat täytyi yhdistää päähaaraan. Kun avoimet haarat olivat yhdistetty päähaaraan, päähaarasta tehtiin uusi kehityshaara. Uuteen haaraan lisättiin hakemisto, joka sisälsi kaiken tarvittavan projektin uudelleen järjestelemiseen. Hakemisto sisälsi kuvien 8, 9 ja 10 skriptit. Lisäksi hakemistossa oli uudet versiot sovelluksen julkaisemiseen tarvittavista tiedostoista. Järjestelemiseen tehdyt skriptit ajettiin, muutokset katselmointiin ja haara yhdistettiin päähaaraan. Tämän jälkeen tiimi pystyi jatkamaan sovelluksen kehittämistä uudella rakenteella.

Yksi kehittäjistä yhdisti onnistuneesti uuden päähaaran omaan kehityshaaraansa, vaikka siinä ei oltu ajettu uudelleen järjestelyn skriptejä. Kyseessä oli siis vanhan rakenteen omaava haara. Konflikteja tuli erittäin vähän. Git-työkalu ei merkannut siirrettyjä tiedostoja poistetuiksi vaan uudelleennimesi ne niiden uusien sijaintien perusteella.

Viimeisenä päivitettiin dokumentaatiota, korjattiin pieniä virheitä ja lisättiin kuvan 11 skripti, joka oli sovelluksen kääntämistä varten Windows-käyttöjärjestelmää käyttäville. Kuva 11 skripti eroaa hieman kuvan 10 skriptistä. Windowsilla symbolinen linkki luodaan itse manuaalisesti. Syy tälle oli se,

että sen luominen tarvitsee admin-oikeudet, joten sen manuaalinen luominen vaikutti parhaalta ratkaisulta. Varsinkin kun symbolinen linkki on luotava vain kerran.

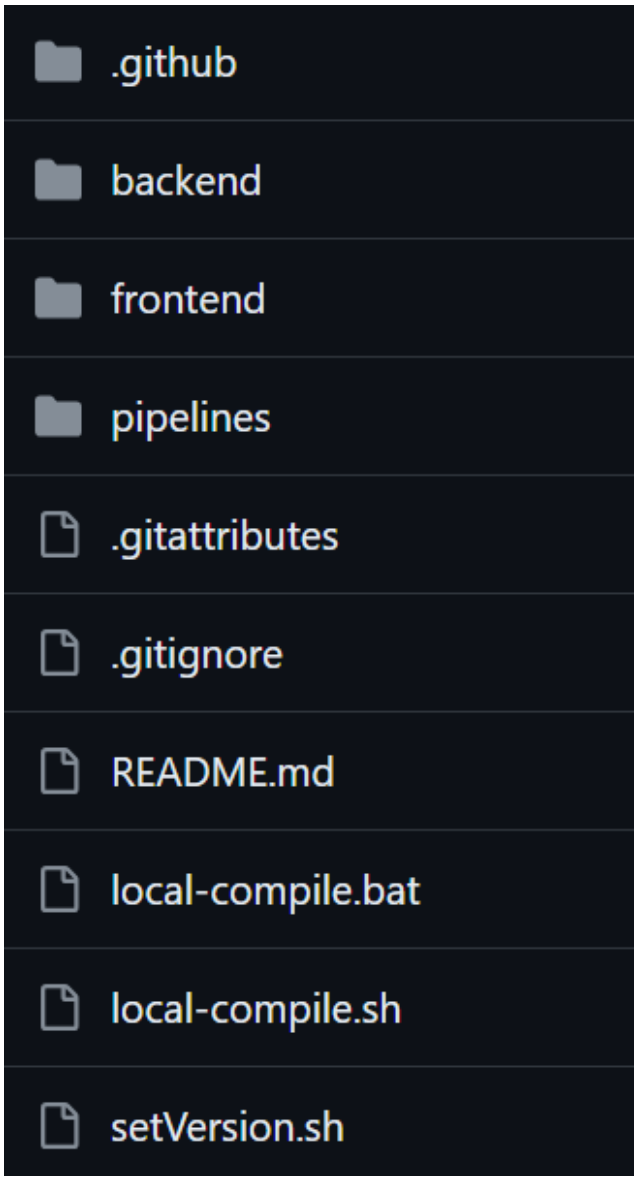
```
@echo off

REM build frontend
pushd frontend
call npm install
call npm run build
popd

REM build backend
pushd backend
if "%1" == "clean" (
  call mvn clean compile
) else (
  call mvn compile
)
popd
```

Kuva 11. local-compile.bat

3.4 Järjestelty lähdekoodi

Tuloksena syntyi siisti ja helpommin lähestyttävä projekti, joka on monorepossa. Alkuperäisessä projektissa juuritasolla oli yli kymmenen hakemistoa. Osa hakemistoista liittyi taustapalveluihin, osa käyttöliittymään ja osa muihin asioihin. Erilaisia tiedostoja oli juuritasolla yli 30. Tiedostoista löytyi konfiguraatiotiedostoja, käyttöliittymän tiedostoja ja muita tiedostoja. Kuvassa 12 on projektin uusi järjestys. Hakemistoja on juuritasolla enää vain neljä kappaletta. Tiedostoja on vain kuusi kappaletta. Tulos mahdollistaa pilvisiirtymän aloittamisen, koska projektin rakenne selkeytyi, ja käyttöliittymä ja taustapalvelu saatiin erotettua toisistaan. Koontiin tarkoitetussa Dockerfile-tiedostossa ei ole enää pitkää luetteloa käyttöliittymän tiedostoista, vaan pelkkä käyttöliittymän hakemisto.

Jatkokehityksen kohteena on erottaa käyttöliittymän ja taustapalvelun koonti ja julkaiseminen erilleen. Kun tämä on toteutettu, pipeline-hakemisto häviää juuritasolta, koska kummallakin palvelulla tulee olemaan omat julkaisemiseen ja koontiin vaaditut tiedostonsa. Ne tullaan sijoittamaan pois omiin hakemistoihinsa.



Kuva 12. Projektin uusi rakenne

4 Pilvipalvelut

Projektin siivoamisen jälkeen on helpompaa lähteä jatkokehittämään sovelluksen käyttöliittymän ja taustapalvelun erottamista. Pidemmällä tähtäimellä tavoitteena on, että palveluiden koonti erotetaan. Lisäksi palveluita tullaan ajamaan eri alustoilla. Tässä luvussa käsitellään pilvipalveluita ja lopuksi toteutetaan prototyyppi sisäänkirjautumissivusta. Päätavoitteena on saada selville, mitä taustapalvelussa täytyy ottaa huomioon, kun palvelut erotetaan sekä tutustua Firebase Hosting-palveluun mahdollisena käyttöliittymän julkaisualustana.

Pilvipalveluita on useita erilaisia. Osa palveluista on selkeästi suunnattu perinteiselle käyttäjälle ja osa organisaatioille. Perinteisille käyttäjille sekä organisaatioille SaaS-tuotteet ovat tuttuja. Muut palvelutyypit kuten PaaS ja IaaS ovat todennäköisesti tutumpia organisaatioille, jotka miettivät, missä ympäristössä omaa tuotettaan haluaa tai missä sitä on parasta ajaa.

Sopivan pilvipalvelun valitseminen tehostaa sovelluskehitystä. Kaikista malleista löytyy hyviä ja huonoja puolia. Pilvipalvelun lisäksi kannattaa kiinnittää huomiota, miten sovellus kootaan ja julkaistaan mahdollisimman helposti ja luotettavasti.

4.1 Pilvipalveluiden historia

Internetin alkuaikoina web-sovellusten ajamista ja julkaisemista varten täytyi hankkia fyysiset palvelimet ja ylläpitää niitä. IT-tiimien piti asentaa palvelimille tarvittavat käyttöjärjestelmät, valmistella ympäristö sovellukselle ja julkaista sovellus näiden päälle. Lähestymistavassa oli useita ongelmia. Palvelimet olivat usein alikäytettyjä, koska niiden koko tehoa ei tarvittu. Lisäksi asennus ja ylläpitokulut olivat korkeat. (Singh & Kehoe 2022, luku 1.)

Virtualisointi kehitettiin, jotta fyysisiä palvelimia voisi hyödyntää tehokkaammin. Prosessoreita ja muistia voitaisiin jakaa. Tämä ratkaisi useita ongelmia, mutta laitteisto täytyi edelleen omistaa sovelluksen julkaisemista varten. Usein tähän liittyi oman konesalin ylläpitäminen. (Singh & Kehoe 2022, luku 1.)

Ongelmat virtualisoinnissa johtivat IaaS-mallin (Infrastructure as a Service) kehittymiseen. IaaS-mallia seurasi PaaS-malli (Platform as a Service). PaaS-malli puolestaan johti SaaS-mallin (Software as a Service) kehittymiseen. (Singh & Kehoe 2022, luku 1.)

4.2 IaaS (Infrastructure as a Service)

Iaas-mallissa palvelimet omistaa kolmas osapuoli. Yrityksen ei siis tarvitse itse huolehtia sovelluksen alla olevasta infrastruktuurista. (Singh & Kehoe 2022, luku 1.) Malli mahdollistaa on-demand (tarpeen vaatiessa) pääsyn resursseihin. Näitä resursseja ovat esimerkiksi palvelimet, tallennustila ja virtualisointi. Malli poistaa yrityksen tarpeen infrastruktuurin hankkimiseen, konfigurointiin ja ylläpitoon (ohjelmistoja lukuun ottamatta). Maksaminen perustuu käytön määrään. IaaS-mallissa vuokrataan pääsy pilvessä olevaan infrastruktuuriin. Pilvipalveluntarjoaja on vastuussa infrastruktuurin ylläpitämisestä. Tilaaja on vastuussa ohjelmiston asentamisesta, konfiguroinnista ja ylläpitämisestä. (Google Cloud s.a. a.)

IaaS-malli tarjoaa dynaamista skaalautuvuutta. Resursseja voidaan automaattisesta skaalata ylöspäin tai alaspäin riippuen sovelluksen tarpeesta. Jos asiakas tarvitsee lisää resursseja, niitä voi saada heti. Usein niiden määrässä on kuitenkin jokin annettu raja. Dynaamiseen skaalautuvuuteen liittyy myös eritasoiset palvelutasot. Sopimus tilaajan ja palveluntarjoajan välillä sisältää usein lupauksen tietystä palvelutasosta. Voidaan esimerkiksi sopia, että resurssien täytyy olla saatavilla 99,999 % ajasta ja, että resurssit skaalautuvat dynaamisesti ylöspäin, jos yli 80 % resursseista ovat käytössä. (Manvi & Shyam 2021, luku 5.1.)

Taulukossa 3 on esitelty hyötyjä ja haittoja IaaS-mallista. Hyödyt ja haitat ovat myös riippuvaisia organisaation tarpeista. Hallinnan korkea taso on hyöty sellaisiin tilanteisiin, joissa halutaan tai täytyy ylläpitää tiettyjä ohjelmistoja. IaaS-malli mahdollistaa sen, että asiat voi tehdä juuri haluamallaan tavalla. Tämä mahdollistaa esimerkiksi vapaan valinnan, mikä käyttöjärjestelmä palvelimella on ja miten se on suojattu.

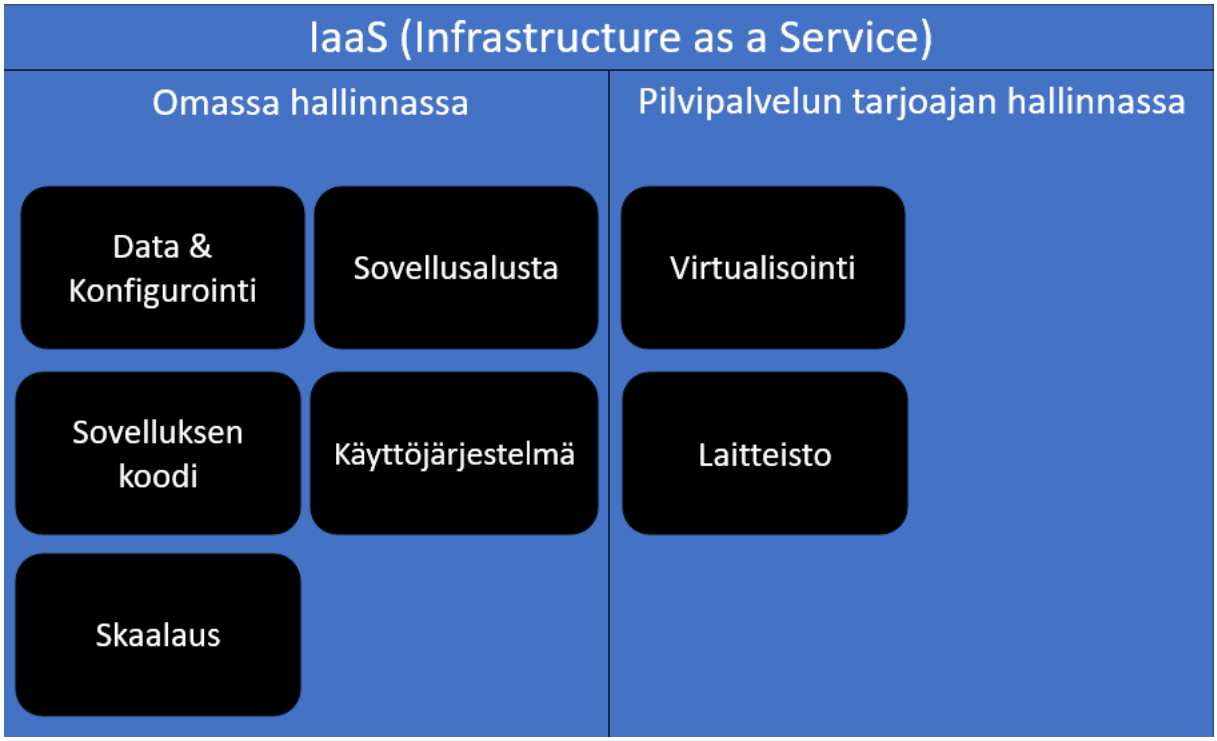
Taulukon haitoista löytyy edellä mainitun hyödyn kääntöpuoli. Jos valitaan tämä malli, palvelimen ylläpitäminen tuo lisää työtä. Pitää muun muassa varmistaa, että käyttöjärjestelmää päivitetään. Tiimin täytyy suojata palvelinta, joka vie aikaa ja resursseja.

Taulukko 3. IaaS-mallin hyötyjä ja haittoja (Google Cloud s.a. b)

Hyödyt	**Haitat**
Muihin malleihin verrattuina korkein taso infrastruktuurin hallinnassa.	Vastuu omasta datasta ja sen palauttamisesta.
Tarpeen mukainen (on-demand) skaalautuvuus.	Vaatii konfigurointia ja ylläpitämistä.
Järjestelmä ei kaadu yhden osan tai komponentin vikatilaan.	Pilvipohjaisessa infrastruktuurissa voi olla vaikeuksia suojata vanhoja sovelluksia.

Kuvassa 13 on mallinnettu, miten hallinta jakautuu IaaS-mallissa pilvipalvelun tarjoajan sekä käyttäjän välillä. Kuvasta huomaa, että käyttäjällä on paljon asioita omassa hallinnassaan. Omassa

hallinnassa on data ja konfigurointi, sovellusalusta, sovelluksen koodi, käyttöjärjestelmä ja skaalaus, kun taas pilvipalvelun tarjoajan hallinnassa on virtualisointi ja laitteisto (Vergadia s.a.).



Kuva 13. IaaS-mallin hallinta käyttäjän ja palveluntarjoajan välillä (mukaillen Vergadia s.a.)

4.3 PaaS (Platform as a Service)

PaaS sisältää kaiken sovelluksen koontiin, ajamiseen ja sovellusten hallitsemiseen. Se tarjoaa palvelimet, käyttöjärjestelmät, varastoinnin ja muita ominaisuuksia. Tiimin täytyy ostaa maksa-kun-käytät pääsy kaikkeen, mitä he tarvitsevat sovelluksen kehittämiseen. Kehittäjät voivat keskittyä sovelluksen kehittämiseen. (Google Cloud s.a. c.) Malli on suunniteltu tarjoamaan web-sovelluksen kehityskaaren kaikki kohdat. Kohtia ovat sovelluksen koonti, testaaminen, julkaiseminen, hallitseminen ja päivittäminen. (Microsoft Azure s.a. a.)

Tämän tyyppisillä malleilla on ominaista niiden tarjoama korkeampi abstraktiotaso. Verrattain edellä mainittuun IaaS-malliin, jossa käyttäjät saavat pääsyn fyysiseen tai virtuaaliseen infrastruktuuriin, PaaS-mallissa käyttäjä voi julkaista hänen sovelluksensa loputtomalta tuntuvaan resurssialtaaseen. Tämän avulla käyttäjältä häviää näkyvistä kaikki monimutkaisuus, joka liittyy sovelluksen julkaisemiseen ja infrastruktuurin konfiguroimiseen. (Manvi & Shyam 2021, luku 6.7.)

Taulukossa 4 on esitetty PaaS-mallin hyötyjä ja haittoja. Taulukosta löytyvät hyödyt voivat olla haittoja toiselle organisaatiolle, kun taas toiselle haitat voivat olla hyötyjä. Mallin valitsemisessa onkin

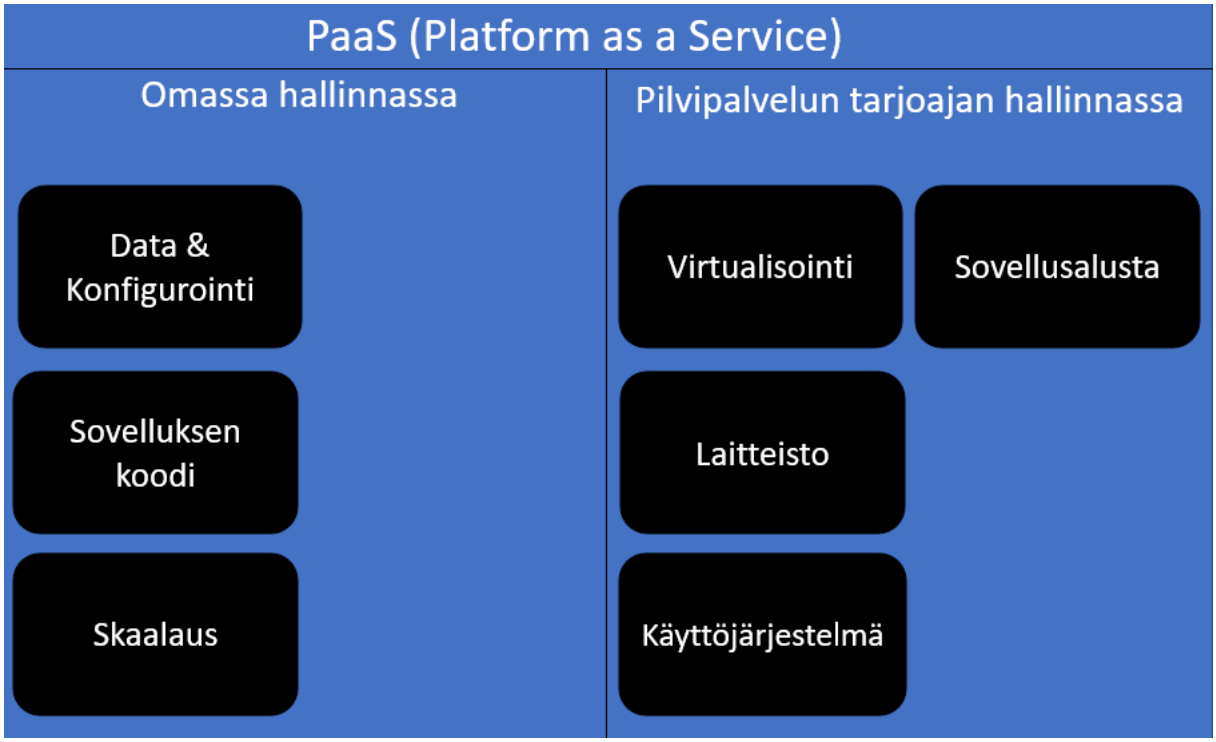
tärkeää tiedostaa, kuinka paljon haluaa jättää itselleen valtaa ja mahdollisuuksia kustomointiin. Toisaalta tämä vapaus lisää myös työmäärää.

Taulukko 4. PaaS-mallin hyötyjä ja haittoja (Google Cloud s.a. b)

Hyödyt	**Haitat**
Palveluntarjoaja on vastuussa infrastruktuurin turvallisuudesta ja ylläpidosta.	Pienempi hallinta infrastruktuuriin.
Skaalautuvuus tarpeen mukaan.	Kustomointi on rajatumpaa.
Välitön pääsy helppokäyttöiseen kehitysympäristöön.	Mahdollinen toimittajaloukku. Tämä on riippuvainen myös pilvipalvelun tarjoajasta.

PaaS sopii hyvin tilanteeseen, jossa halutaan ulkoistaa useita sovelluksen julkaisemiseen ja ajamiseen liittyviä vaiheita. PaaS-malli rajaa hieman, miten tiimi pystyy hallitsemaan ja konfiguroimaan ympäristöä, missä sovellusta ajetaan. Toisaalta tämä malli tuo vapauden palvelimien ylläpitämisestä, konfiguroimisesta ja muista ylläpitoon liittyvistä tehtävistä. Vapautuvan ajan voi käyttää sovelluksen kehittämiseen.

Kuvassa 14 on esitetty, miten hallitseminen jakautuu PaaS-mallissa käyttäjän sekä pilvipalvelun tarjoajan välillä. Edellä mainittuun IaaS-malliin verrattuna, ovat käyttöjärjestelmä sekä sovellusalusta siirtyneet palveluntarjoajan hallinnan alle.



Kuva 14. PaaS-mallin hallinta käyttäjän ja palveluntarjoajan välillä (mukaillen Vergadia s.a.)

4.4 SaaS (Software as a Service)

SaaS-malli on jokaiselle, joka käyttää internettiä tuttu, vaikka sitä ei itse tietäisi. Useat käyttävät SaaS-palvelua päivittäin. Hyviä esimerkkejä SaaS-alustoista ovat Gmail, Outlook, Slack ja Zeroni. Nämä sovellukset tavoittavat useita käyttäjiä, jotka pääsevät käyttämään sovellusta selaimen kautta.

SaaS-palvelussa käyttäjät pääsevät valmiiseen pilvipohjaiseen sovellukseen käsiksi. Sovellusta hallinnoi kokonaan palveluntarjoaja. Palveluntarjoaja tarjoaa käyttövalmiin tuotteen, jota päivitetään ja ylläpidetään. Useimmat SaaS-tuotteet ovat web-sovelluksia, jolloin käyttäjän ei tarvitse ladata mitään omalle laitteelleen. (Google Cloud s.a. b.) Malli tarjoaa maksa-kun-käytät-tyyppisen ratkaisun. Kaikki infrastruktuuri, sovelluksen lähdekoodi ja sovelluksen data ovat pilvipalvelun tarjoajan omissa konesaleissa. (Microsoft Azure s.a. b.)

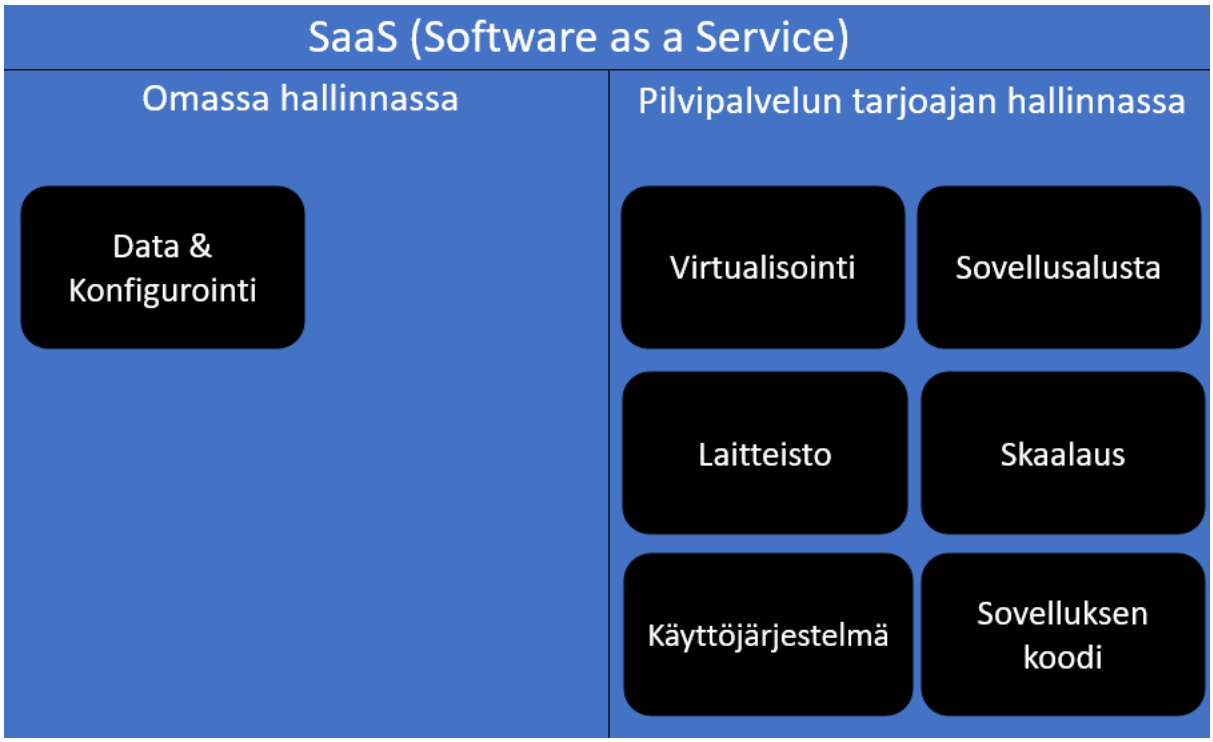
SaaS-tuotteessa käyttäjä voi konfiguroida tuotteen ulkoasua ja toiminnallisuutta itselleen sopivaksi. Nämä voivat tarkoittaa esimerkiksi kustomoitua logoa ja kustomoitua värimaailmaa. Kustomoinnilla on rajansa. Käyttäjät eivät esimerkiksi pysty välttämättä muuttamaan sovelluksen sivun pohjaa, ellei tällaista tukea ole tuotteeseen suunniteltu. SaaS-tuotteiden päivittämissykli on usein tiheämpää. Päivityksiä saattaa tulla viikoittain. Tiheän syklin mahdollistaa se, että sovellusta ajetaan keskitetysti ja päivitykset tapahtuvat palveluntarjoajan päätöksellä ja toteutuksella. Asiakkaan ei tarvitse itse päivittää mitään. (Manvi & Shyam 2021, luku 6.2.)

Taulukossa 5 on kuvattu SaaS-palvelun hyötyjä ja haittoja. Sekä taulukosta 5, että kuvasta 15 näkee, miten hallinta on siirtynyt palveluntarjoajalle. Malli tarjoaa käyttäjille suhteellisen huolettoman mahdollisuuden päästä käsiksi erilaisiin tuotteisiin. Tässä mallissa kustomointi juuri omiin tarpeisiin on rajallista. Malli ei välttämättä sovi organisaatioille tai yksityishenkilöille, joiden tarve on todella omalaatuista ja vaikeasti tuotteistettavaa.

Taulukko 5. SaaS-mallin hyötyjä ja haittoja (Google Cloud s.a. b)

Hyödyt	**Haitat**
Helppo ottaa käyttöön.	Ei mahdollisuutta vaikuttaa infrastruktuuriin tai tietoturvaan.
Palveluntarjoaja ylläpitää ja hallitsee kaikkea, kuten laitteiston ja sovelluksen.	Mahdollinen toimittajaloukku. Tämä on riippuvainen myös pilvipalvelun tarjoajasta.
Tuotetta voi käyttää millä laitteella tahansa, kunhan laitteessa on internetyhteys.	Integraatio jo olemassa olevien työkalujen ja sovellusten kanssa voi olla haastavaa.

Kuvassa 15 huomaa, miten käyttäjän hallintaan on jäänyt enää data ja palvelun jonkin tasoinen konfigurointi itselleen sopivaksi. Hallinnan vähenemisen kustannuksella saa ulkoistettua melkein kaiken palveluntarjoajalle.



Kuva 15. SaaS-mallin hallinta käyttäjän ja palveluntarjoajan välillä (mukaillen Vergadia s.a.)

5 Tulevia työkaluja

Zeronin pilvisiirtymään liittyy vahvasti Google Cloud Run sekä Firebase Hosting. Cloud Run perustuu konttiteknologiaan. Zeronin koonnissa sekä julkaisemisessa hyödynnetään jo nyt kontteja. Seuraavaksi esiteltävät asiat ovat olennaisessa roolissa tuotteen jatkokehityksessä ja pilvisiirtymän toteutuksessa.

Google Cloud Run palveluun liittyvä oleellinen tekniikka on kontit. Kontti on eristetty ympäristö, joka sisältää kaiken, mitä koodi tarvitsee toimiakseen käyttöjärjestelmään asti. (Docker Inc. s.a. b.) Docker-kontti on kevyt, itsenäinen ja ajettava paketti ohjelmasta. Se sisältää kaiken, mitä tarvitaan sovelluksen ajamiseen. (Docker Inc. s.a. a.)

5.1 Google Cloud Run

Google Cloud Run on konttipohjaiseen kehitykseen tarkoitettu PaaS-ratkaisu (Google Cloud s.a. c). Palvelussa voidaan ajaa kontteja Googlen skaalautuvan infrastruktuurin päällä. Ohjelmointikielellä ei ole merkitystä. Jos sovelluksesta pystyy luomaan konttikuvan, sen voi julkaista Cloud Run-palveluun. (Google Cloud s.a. d.)

Palvelun avulla kehittäjien ei tarvitse hallita infrastruktuuria, vaan ajan voi käyttää sovelluksen kehittämiseen. Koodia voi ajaa, joko palveluna tai tehtävänä. Cloud Run Services on käytössä silloin, kun koodia vastaa verkkopyyntöihin tai tapahtumiin. Cloud Run Jobs on taas käytössä, kun koodi tekee tehtävänsä ajoitetusti ja lopettaa tehtävän suoritettuaan. Cloud Run Services sopii hyvin nettisivuille ja nettisovelluksille, ohjelmistorajapinnoille ja mikropalveluille. Cloud Run Jobs on sopiva skriptien ja ajastettujen töiden ajamiseen. (Google Cloud s.a. d.)

5.2 Firebase Hosting

Firebase Hosting on palvelu, joka mahdollistaa käyttäjilleen verkkosovellusten, staattisen ja dynaamisen sisällön, sekä mikropalvelujen ajamisen ja julkaisemisen helposti. Firebase CLI-työkalun avulla sovelluksen käynnistäminen ja julkaiseminen on helppoa. (Firebase s.a. a.)

Konfigurointi tapahtuu firebase.json-tiedostossa. Konfiguroinnissa voi muun muassa määritellä, mitkä tiedostot lokaalista hakemistosta julkaistaan, asettamaan uudelleenohjauksia siirretyille tai poistetuille sivuille ja kertomaan, mitä tiedostoja ei oteta julkaisemiseen mukaan. Ainoa pakollinen tieto on konfiguroida, mitkä tiedostot otetaan julkaisemiseen mukaan. (Firebase s.a. b.)

Kuvassa 16 on esimerkki firebase.json-tiedostosta. Public kertoo tiedostot, jotka julkaistaan. Esimerkissä tiedostot sijaitsevat hakemistossa nimeltä dist. Ignore-taulukko sisältää tiedostot, joita ei haluta julkaista. Redirects taulukkoon voi antaa sääntöjä, joiden mukaan uudelleenohjautuminen

tapahtuu. Source on lähde, jonka saamat kutsut ohjautuvat destination kohdassa määriteltyyn kohteeseen. Type on HTTPS vastaus koodi. (Firebase s.a. b.) Koodi 301 tarkoittaa, että pyydetty resurssi on saanut uuden pysyvän URI:n (Fielding, Nottingham & Reschke 2022, luku 15.4.3).

Kuvan esimerkissä kaikki pyynnöt, jotka ovat /blogi osoitteelle, uudelleenohjautuvat osoitteeseen /uusi-blogi.

```
{
    "hosting": {
        "public": "dist",
        "ignore": [
            "firebase.json",
            "**/.*",
            "**/node_modules/**"
        ],
        "redirects": [
            {
                "source": "/blogi",
                "destination": "/uusi-blogi",
                "type": 301
            }
        ]
    }
}
```

Kuva 16. Esimerkki firebase.json-tiedostosta

6 Zeronin pilvisiirtymän lähtölaukaus

Zeronia ajetaan tällä hetkellä IaaS-palvelussa. Tavoitteena on siirtyä kohti PaaS-mallia taustapalvelun osalta. Tähän on valikoitu Google Cloud Run palvelu. Käyttöliittymä tullaan viemää Firebase Hosting palveluun.

Pilvisiirtymä tulee olemaan jatkokehitettävä asia. Tässä osiossa käsitellään pilvisiirtymän syitä sen kautta, mitä hyödyllisiä tekniikoita se tulee mahdollistamaan. Prototyypin avulla saadaan tietoa, miten taustapalvelua tulee muuttaa jatkokehitysvaiheessa.

6.1 Pilvisiirtymän syyt

PaaS-mallissa saadaan ulkoistettua useita eri ylläpitotehtäviä pilvipalveluntarjoajalle. Erityisesti skaalautuvuus koetaan hyötynä. Lisäksi saadaan saavutettavuutta parannettua. Siirtymän pidemmän aikavälin tavoitteena on useamman Zeronin samanaikainen ajaminen. Tämän avulla jatkuva päivittäminen on helpompaa, koska voidaan välttää palvelukatkoksia. Jos esimerkiksi ajossa olisi kolme Zeronia saman aikaisesti, ne voidaan päivittää erikseen. Kun yhtä päivitetään, liikenne ohjataan kahteen muuhun ajossa olevaan Zeroniin.

Useamman Zeronin ajaminen mahdollistaa jatkuvan toimittamisen parantumisen, mikä on SaaS-mallin yksi keskeisistä piirteistä. Päivitystiheyttä voidaan näin lisätä, mikä auttaa mahdollisten virheiden paikantamisen helpottumisen. Kun tuotantoon viedään pienemmissä osissa päivityksiä, voidaan yksittäinen päivitys perua helpommin, kuitenkaan menettämättä muita uusia ominaisuuksia.

6.2 Prototyypin tavoitteet

Seuraava tavoite on luoda prototyyppi sisäänkirjautumissivusta. Prototyyppi luodaan, jotta saamme tietoa taustapalvelun muuttamiseen vaadittavista asioista. Nyt Zeronin taustapalvelua sekä käyttöliittymää ajetaan samassa ympäristössä.

Prototyypin tekeminen ja tiedon saaminen tukee tavoitetta, jossa käyttöliittymä ja taustapalvelu saadaan erotettua toisistaan. Tavoitteena on, että prototyyppi tekee pyyntöjä taustapalveluun. Lisäksi tutustutaan Firebase Hosting-palveluun.

6.3 Suunnitelma prototyypin toteutuksesta

Prototyyppi tulee sisältämään todella yksinkertaisen sisäänkirjautumissivun sekä sovelluksen etusivun. Sisäänkirjautumissivulla tulee olemaan kaksi tekstikenttää ja painike, jolla voi kirjautua sisälle. Onnistuneet kirjautumisen jälkeen sivulle tulostuu testidataa, joka on haettu taustapalvelusta. Testidatan lisäksi sivulla on uloskirjautumiseen tarkoitettu painike.

Prototyyppi tullaan toteuttamaan React-sovelluksena. Sovellukselle luodaan uusi täysin tyhjä React sovellus. Sovelluksen luomiseen tullaan käyttämään https://create-react-app.dev/docs/getting-started sivulta löytyvää mahdollisuutta luoda sovellus helposti ja ilman konfigurointia. Kieleksi valitaan TypeScript. Lisäksi asennetaan Firebase CLI-työkalu. Prototyypissä ei tehdä asioita alan parhaiden käytäntöjen mukaan, koska sillä ei ole tavoitteiden saavuttamisen suhteen merkitystä.

6.4 Prototyypin toteutus

Prototyypin toteutus oli hyvin yksinkertainen ja nopea. Toteutus aloitettiin luomalla uusi React-sovellus. Sovellukseen tuli kaksi näkymää. Näkymät olivat sisäänkirjautumissivu, sekä sovelluksen etusivu. Lisäksi kummallekin näkymälle tuli omat service-tiedostot, joissa tapahtui pyyntöjen teko taustapalveluun.

Ensimmäisillä yrityksillä lisäsin prototyypin package.json-tiedostoon proxy-konfiguraation. Konfiguraation avulla kehitysympäristössä tuntemattomat pyynnöt välitetään konfiguraatiosta löytyvään osoitteeseen. Esimerkiksi jos proxyn arvona on http://localhost:4000, pyyntö fetch('/api/foo') välittyy osoitteeseen http://localhost:4000/api/foo. Tämän asetuksen avulla voidaan välttää CORS-ongelmat kehitysympäristössä. (Create React App s.a.). Tämän avulla pystyin varmistumaan, että pyynnöissä ei ole virheellinen osoite ja, että ne toimivat kuten kuuluu.

Ongelmaksi muodostui se, että taustapalvelussa oli uudelleenohjauksia. Uudelleenohjaukset eivät toimineet enää tämän prototyypin kanssa, koska palveluita ajettiin eri porteista. Muutin taustapalvelusta uudelleenohjautumista. Taustapalvelusta täytyi löytää oikea kohta, jossa sisäänkirjautumislogiikka oli toteutettu. Nyt onnistuneet sisäänkirjautumisen jälkeen, taustapalvelu ohjasi takaisin prototyypin osoitteeseen.

Viimeisenä tutustuttiin Firebase Hosting-konfigurointiin. Asensin Firebase CLI-työkalun. Loin työkalulla uuden projektin. Projektin luonnin yhteydessä prototyypin juureen tuli firebase.json-tiedosto. Tiedostoon lisäsin hakemiston, johon sovelluksen koonti tulee. Lisäksi lisäsin rewrites-säännön. Prototyypissä onnistuneen sisäänkirjautumisen jälkeen taustapalvelu ohjasi osoitteeseen http://localhost:3000/dashboard. Tätä osoitetta ei ole olemassa. Sitä käytettiin ainoastaan tapana, miten renderöidään eri näkymät. Jos osoitepalkissa oli sana dashboard, renderöitiin Dashboard.tsx-komponentti. Tietojen haku ei kuitenkaan onnistunut ilman onnistunutta sisäänkirjautumista. Koska osoitteesta ei löydy resursseja, rewrites-sääntö ohjaa index.html-tiedostoon. Kuvassa 17 on prototyypin firebase.json-tiedosto.

Ennen Hosting kokeilua sovellus käynnistettiin aina npm start komennolla. Hosting kokeilua varten sovellus koottiin komennolla npm run build. Komento luo optimoidun tuotantoversion sovelluksesta.

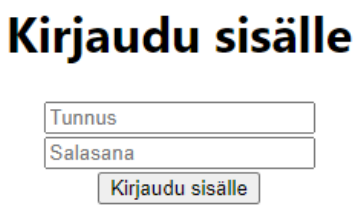
Lopuksi annettiin komento firebase emulators:start. Komento käynnistää konfiguraation perusteella sovelluksen osoitteeseen http://localhost:5000.

```
{
  "hosting": {
    "public": "build",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}
```

Kuva 17. Prototyypin firebase.json-tiedosto

6.5 Prototyyppi

Prototyyppi oli ulkonäöltään ja toteutukseltaan hyvin yksinkertainen, niin kuin oli suunniteltu. Kuvassa 18 on prototyypin sisäänkirjautumissivu. Sivu sisältää tekstikentät sekä käyttäjätunnukselle, että salasanalle. Lisäksi on painike, jolla sisäänkirjautuminen suoritetaan.



Kuva 18. Prototyypin sisäänkirjautumissivu

Kuvassa 19 on näkymä, jossa sisäänkirjautuminen on onnistunut. Onnistuneen sisäänkirjautumisen jälkeen on haettu testidataa taustapalvelusta, sekä tulostettu ne.

Kuva 19. Prototyypin etusivu onnistuneen sisäänkirjautumisen jälkeen

6.6 Tulokset

Prototyypin yksinkertaisuudesta huolimatta saatiin tärkeää tietoa siitä, miten taustapalvelua täytyy jatkokehittää. Koko sisäänkirjautuminen täytyy suunnitella uudelleen sekä taustapalveluun, että käyttöliittymään. Taustapalvelun ei pitäisi jatkossa tehdä lähes ollenkaan uudelleenohjauksia. Sen pitäisi toimia pelkästään rajapintana, josta haetaan pyynnöillä dataa. Muutokset tulevat vaikuttamaan myös pyyntöjen oikeuksien tarkastamiseen.

Prototyyppi antoi hyvän lähtökohdan lähteä jatkokehittämään sekä taustapalvelua, että käyttöliittymää. Jatkokehitysvaiheessa eteen tulee varmasti uusia ongelmia vastaa, jotka täytyy ratkoa. Näiden tuloksien avulla pystytään kuitenkin osoittamaan, mistä lähteä liikkeelle.

7 Pohdinta

Opinnäytetyön tavoitteena oli valita toimeksiantajalle sopiva repositoriomalli, erottaa tiedostotasolla käyttöliittymä ja taustapalvelu toisistaan sekä aloittaa pilvisiirtymän valmisteleminen. Pilvisiirtymän valmistelemisen tavoitteena oli saada tietoa taustapalvelun jatkokehityskohteista. Mielestäni tavoitteisiin päästiin.

Opinnäytetyön ensimmäisessä teoriaosuudessa käsiteltiin vaihtoehtoja repositoriomallille. Teoriaosuuden avulla opin paljon uutta eri vaihtoehdoista, miten lähdekoodia voi säilyttää. Suurin oivallus oli, että niin kuin monessa muussakin asiassa, ei ole olemassa täydellistä ratkaisua. Vaihtoehdoista täytyy löytää itselleen tai tiimilleen paras ratkaisu.

Ennen ensimmäistä toiminnallista osiota olisin voinut tutustua enemmän Maven-projektien rakenteeseen. Jos olisin tutustunut rakenteeseen paremmin, toimeksiantajan projektin rakenteen ymmärtäminen olisi ollut helpompaa. Uskon, että tämä olisi nopeuttanut ensimmäistä toiminnallista osuutta.

Ensimmäisessä toiminnallisessa osiossa toteutettiin valittu repositoriomalli sekä erotettiin taustapalvelu ja käyttöliittymä toisistaan tiedostotasolla. Tässä osiossa opin lisää skripteistä sekä Maven-projektin rakenteesta. Tuloksena oli siisti projektin rakenne, joka on tuotekehitystiimillä käytössä. Yksi keskeinen tavoite tässä osiossa oli siistityn rakenteen käyttöönotto ilman suurempia ongelmia. Käyttöönotto ei aiheuttanut uusia ongelmia eikä mikään ohjelmiston osa hajonnut, joten tavoitteeseen päästiin. Onnistumisen takasi hyvä kommunikaatio tiimin sisällä sekä riippuvuussuhteiden löytäminen ja niiden oikeanlainen päivittäminen. Nykyisellä projektin rakenteella tuotteen jatkokehittäminen kohti pilvisiirtymää on helpompaa, koska esimerkiksi sovelluksen koontiin tarvittavat tiedostot ovat helpommin saatavilla.

Toisessa teoriaosuudessa käsiteltiin pilvipalvelumalleja. Osuuden avulla ymmärsin laajemmin eri mallien eroja ja hyötyjä. Pilvipalvelumallin valinnassa on tärkeä löytää omiin tarpeisiin sopiva malli. Saamalla joitain hyötyjä, saattaa joutua luopumaan jostain muusta. Teoriaosuuden jälkeen perustellaan toimeksiantajan siirtymistä IaaS-mallista kohti PaaS-mallia. IaaS-mallissa on enemmän mahdollisuuksia itse vaikuttaa asioihin, mutta PaaS-mallin tuomat hyödyt painavat enemmän vaakakupissa. Osion avulla ymmärsin, mitä hyötyjä PaaS-malli voi tuoda Zeroni-tuotteen hallintaan ja ylläpitoon.

Toiminnallisessa osiossa toteutettiin yksinkertainen prototyyppi sisäänkirjautumissivusta. Toiminnallisen osion avulla tiimi sai tietoa taustapalvelun jatkokehitystarpeista, joten osio oli onnistunut.

Tulosten avulla voi nimetä kehitettäviä asioita, joista yksi on koko sisäänkirjautumisen uudelleen suunnitteleminen.

Työtä tehdessä olen huomannut, kuinka tärkeää suunnitteleminen on. Tämä on yksi selkeä kehityskohta. Opinnäytetyötä suunnitellessani olisin voinut käyttää enemmän aikaa sisällön suunnittelemiseen. Samalla tavalla ennen toiminnallisia osioita, olisin voinut suunnitella paremmin, mitkä ovat osion tavoitteet ja miten tavoitteisiin päästään. Alan usein tekemään liian innokkaasti ja tämä saattaa jopa hidastaa lopputulokseen pääsemistä.

Työssä päästiin tavoitteisiin ja opin paljon uutta. Jatkokehitykseen jää kuitenkin myös paljon asioita. Tiimi voisi siistiä jatkokehityksenä projektista turhia ja käyttämättömiä tiedostoja pois. Vanhoja tiedostoja kertyy helposti projekteihin ja niitä harvemmin uskalletaan poistaa, koska ei olla varmoja tarvitaanko niitä. Täytyy myös kehittää tapoja, joilla varmistetaan, että projekti pysyy siistinä jatkossakin. Suurimmaksi jatkokehityksen kohteeksi jää pilvisiirtymän edistäminen.

Lähteet

Atlassian s.a. a. What is version control? Luettavissa: https://www.atlassian.com/git/tutorials/what-is-version-control. Luettu: 12.12.2023.

Create React App. s.a. Proxying API Requests in Development. Luettavissa: https://create-react-app.dev/docs/proxying-api-requests-in-development/. Luettu: 16.3.2024.

Chacon, S. & Straub, B. 2014. Pro Git. 2. painos. Apress. Berkeley. E-kirja. Luettu: 4.2.2024.

Davis, A. 2021. Bootstrapping Microservices with Docker, Kubernetes, and Terraform. Manning Publications. New York. E-kirja. Luettu: 18.3.2024.

Docker Inc. s.a. a. Use containers to Build, Share and Run your applications. Luettavissa: https://www.docker.com/resources/what-container/. Luettu: 11.3.2024.

Docker Inc. s.a. b. What is a container? Luettavissa: https://docs.docker.com/guides/walkthroughs/what-is-a-container/. Luettu: 11.3.2024.

Fielding, R., Nottingham, M. & Reschke, J. 2022. HTTP Semantics. RFC 9110. Luettu: 13.3.2024.

Firebase s.a. a. Firebase Hosting. Luettavissa: https://firebase.google.com/docs/hosting. Luettu: 11.3.2024.

Firebase s.a. b. Configure Hosting behavior. Luettavissa: https://firebase.google.com/docs/hosting/full-config. Luettu: 13.3.2024.

GitHub s.a. a. Features. Luettavissa: https://github.com/features. Luettu: 18.12.2023.

GitHub s.a. b. Hello World. Luettavissa: https://docs.github.com/en/get-started/quickstart/hello-world. Luettu: 18.12.2023.

GitHub s.a. c. mateodelnorte/meta. Luettavissa: https://github.com/mateodelnorte/meta. Luettu: 22.1.2024.

GitHub s.a. d. mklabs/tabtab. Luettavissa: https://github.com/mklabs/tabtab. Luettu: 22.1.2024.

Google Cloud s.a. a. What is IaaS? Luettavissa: https://cloud.google.com/learn/what-is-iaas. Luettu: 18.2.2024.

Google Cloud s.a. b. PaaS vs. IaaS vs. SaaS vs. CaaS: How are they different? Luettavissa: https://cloud.google.com/learn/paas-vs-iaas-vs-saas. Luettu: 18.2.2024.

Google Cloud s.a. c. What is Platform as a Service (PaaS)? Luettavissa: https://cloud.google.com/learn/what-is-paas. Luettu: 26.2.2024.

Google Cloud s.a. d. What is Cloud Run. Luettavissa: https://cloud.google.com/run/docs/overview/what-is-cloud-run. Luettu: 11.3.2024.

Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E.K., Winter, C. & Murphy-Hill, E. 2018. Advantages and disadvantages of a monolithic repository: a case study at google. ACM Digital Library, In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, s. 225–234. Luettu: 19.12.2023.

Kaufmann, M. 2022. Accelerate DevOps with GitHub: Enhance Software Delivery Performance with GitHub Issues, Projects, Actions, and Advanced Security. Packt Publishing. Birmingham. E-kirja. Luettu: 17.3.2024.

Kumari, S. 2015. Linux shell scripting essentials: Learn shell scripting to solve complex shell-related problems and efficiently automate your day-to-day tasks. Packt Publishing. Birmingham. E-kirja. Luettu: 4.3.2024.

Manvi, S. & Shyam, G. 2021. Cloud Computing. CRC Press. Boca Raton. E-kirja. Luettu: 18.2.2024.

Microsoft Azure s.a. a. What is PaaS? Luettavissa: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-paas. Luettu: 26.2.2024.

Microsoft Azure s.a. b. What is SaaS? Luettavissa: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-saas. Luettu: 27.2.2024.

Potvin, R. & Levenberg, J. Why Google stores billions of lines of code in a single repository. 2016. Communications of the ACM, 59, 7, s. 78–87. Luettu: 19.12.2023.

Roldán, C. 2023. React 18 Design Patterns and Best Practices. 4. painos. Packt Publishing. Birmingham. E-kirja. Luettu: 19.12.2023.

Scholl, B., Jausovec, P. & Swanson, T. 2019. Cloud Native. O'Reilly Media, Inc. Sebastopol. E-kirja. Luettu: 2.1.2024.

Singh, N. & Kehoe, M. 2022. Cloud Native Infrastructure with Azure. O'Reilly Media Inc. Sebastopol. E-kirja. Luettu: 18.2.2024.

snyk. s.a. Open Source Security Explained. Luettavissa: https://snyk.io/series/open-source-security/. Luettu: 22.1.2024.

The Apache Software Foundation 2023. Introduction to the Standard Directory Layout. Luettavissa: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html. Luettu: 10.2.2024.

Ubuntu Manpage s.a. a. set. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/set.1posix.html. Luettu: 3.3.2024.

Ubuntu Manpage s.a. b. mkdir. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/mkdir.1.html. Luettu: 3.3.2024.

Ubuntu Manpage s.a. c. cp. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/cp.1.html. Luettu: 4.3.2024.

Ubuntu Manpage s.a. d. mv. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/mv.1.html. Luettu: 4.3.2024.

Ubuntu Manpage s.a. e. echo. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/echo.1.html. Luettu: 4.3.2024.

Ubuntu Manpage s.a. f. find. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/find.1.html. Luettu: 4.3.2024.

Ubuntu Manpage s.a. g. sed. Luettavissa: https://manpages.ubuntu.com/manpages/trusty/man1/sed.1.html. Luettu: 4.3.2024.

Vergadia, P. s.a. What-is-cloud? Luettavissa: https://www.thecloudgirl.dev/compute/what-is-cloud. Luettu: 20.2.2024.

Vinci, R. 2023. Building Micro Frontends with React 18: Develop and Deploy Scalable Applications Using Micro Frontend Strategies. Packt Publishing. Birmingham. E-kirja. Luettu: 2.1.2024.