Hyvän ohjelmistoarkkitehtuurin piirteet

Teemu Kosonen

Haaga-Helia ammattikorkeakoulu

AMK-opinnäytetyö

2021

IT-tradenomin tutkinto

Tiivistelmä

Tekijä(t)
Teemu Kosonen

Tutkinto
IT-tradenomi

Raportin/Opinnäytetyön nimi
Hyvän ohjelmistoarkkitehtuurin piirteet

Sivu- ja liitesivumäärä
21 + 5

Opinnäytetyö tutkii hyviä ohjelmistoarkkitehtuurin piirteitä sekä periaatteita. Opinnäytetyössä seurataan myös tekijän toteuttaman ohjelmistoprojektin kehitystä. Aihe syntyi tekijän henkilökohtaisesta kiinnostuksesta ohjelmistoarkkitehtuuriin.

Tavoitteena opinnäytetyölle oli tutkia sekä selvittää hyviä ohjelmistoarkkitehtuurin piirteitä. Lisäksi epävirallisena tavoitteena oli toteuttaa tutkimuksen ohella ohjelmistoprojekti, jonka arkkitehtuuria myös tekijä tutki. Ohjelmistoprojektin tavoite oli toimiva prototyyppi.

Arkkitehtuuri ohjelmistokehityksessä on erittäin tärkeää. Tutkimuksessa kävi ilmi, että huonosti toteutettu arkkitehtuuri voi johtaa lähdekoodin rappeutumiseen sekä liiketoiminnan kaatumiseen.

Projektin tavoitteena oli toteuttaa tikettijärjestelmä, jossa ajatuksena oli, että tiketit jakautuvat automaattisesti järjestelmän käyttäjien välillä hyödyntäen tekijän aikaansaamaa algoritmia.

Tutkimuksessa selvinneiden tulosten mukaan ohjelmistojen arkkitehtuurit on parasta jakaa riittävän pieniin kokonaisuuksiin helpottuvan ylläpidon vuoksi.

Tekijä oppi uusia huomioitavia asioita ohjelmistoarkkitehtuuriin liittyen. Tekijä myös ymmärsi, että huonosti toteutettu arkkitehtuuri vaikuttaa negatiivisesti sovelluksen elinikään ja johtaa pahimmassa tapauksessa liiketoiminnan romahdukseen.

Asiasanat
arkkitehtuuri, ohjelmointi, ohjelmistokehitys

Sisällysluettelo

Johdanto

Ohjelmistokehitys on ollut merkittävässä nousussa nyt jo useamman vuoden. Sovellukset sekä muut ohjelmistot monimutkaistuvat niiden haaliessa lisää toiminnollisuuksia. On tärkeää, että suurille ohjelmille on laadittu perustavanlaatuiset suunnitelmat, varsinkin niiden rakenteelle. Tätä varten ovat arkkitehtuurimallit. Niiden avulla ohjelmisto saadaan noudattamaan hyväksi todettuja arkkitehtuurillisia sekä rakenteellisia käytänteitä, jotka hyödyttävät niin kehittäjiä, kuin myös tuotteen loppukäyttäjiä.

Projektin synty

Tässä tutkimuksessa perehdytään tarkemmin ohjelmistoarkkitehtuuriin etenkin tutkimuksen ohella toteutettavan projektin näkökulmasta. Tutkimuksen aiheen perustana on oma henkilökohtainen kiinnostukseni ohjelmistoarkkitehtuuriin ja siihen miten sellainen toteutetaan laatua tarkkaillen. Projektina lähden toteuttamaan tikettijärjestelmää, jonka idea on jakaa tikettejä järjestelmän käyttäjien kesken, hyödyntäen toteuttamaani jakelualgoritmia.

Merkittävyys

Projektin merkittävyys piilee sen arkipäiväisyydessä. Tämän päivän ohjelmistokehityksessä ns. tikettitaulut ovat jo standardi työarjessa. Haluisinkin yrittää viedä tikettitaulun konseptina vielä askeleen pidemmälle lisäämällä siihen automaatiota manuaalisen työn vähentämiseksi sekä toiminnan suoraviivaistamiseksi.

Yksinkertaisesti selitettynä ajatus siis on, että käyttäjä luo tiketin, jonka jälkeen ohjelmiston oma algoritmi laskelmoi tiketille parhaan mahdollisen ratkojan järjestelmän käyttäjien välillä. Projektin on tarkoitus saada aikaan toimiva prototyyppi, jota olisi sitten mahdollista lähteä tarvittaessa kehittämään eteenpäin käyttökelvolliseksi tuotteeksi yritysmaailmaan.

Tavoitteet

Tutkimuksen päätavoite oppimisnäkökulmasta tarkasteltuna on selvittää, millaista on hyvä ohjelmistoarkkitehtuuri. Tämä tietysti riippuu täysin järjestelmän laajuudesta, sen toiminnan tarkoituksesta sekä sen eri osien lukumäärästä. Kuitenkin pyrin selvittämään mahdollisimman yleispäteviä hyvän arkkitehtuurin ominaisuuksia. Projektin arkkitehtuurin tulee tavoitteellisesti noudattaa ”Three-tier-architecture”:n käytänteitä.

Tapa, jolla lähden tämän projektin arkkitehtuuria lähestymään sekä tarkastelemaan, on oman oppimiseni kannalta varsin mielenkiintoinen. Ensin pyrin toteuttamaan projektin sovelluksen omien tämänhetkisten taitojen mukaisesti, tietysti tarvittaessa hieman nojaten internetin tarjoamaan tietoon. Kun tämä sovelluskokonaisuus saadaan riittävän valmiiksi, lähden tarkastelemaan oman koodini arkkitehtuurillisia ratkaisuja aineistoihin perustuen ja niihin verraten pohdin, miten hyvin osasin sovelluksen toteuttaa. Voisi siis ajatella, että tämä opinnäytetyö on itselleni järjestämä todellinen testi kaikesta korkeakoulun aikana oppimastani aiheeseen liittyvästä teoriasta ja käytännöstä.

Projektin työkalut

Projektin järjestelmä sisältää asiakasohjelman (frontend), palvelimen (backend), sekä tietokannan (database). Asiakasohjelma on sovellus, jonka visuaalisen käyttöliittymän loppukäyttäjät näkevät, ja jota he myös käyttävät. Tietokanta säilyttää sekä ylläpitää järjestelmän datamassaa. Palvelin, etenkin tutkimuksen ohella toteutettavan projektisovelluksen tapauksessa toimii asiakasohjelman ”viestinviejänä”. Asiakasohjelma tullaan toteuttamaan modernia JavaScriptia sekä React-kirjastoa hyödyntäen, koska nämä - etenkin Javascript - ovat miltei nykyajan standardi web-sovelluskehityksessä. Lisäksi tämä on mielestäni kiinnostava teknologia sekä myös soveltuu tähän tehtävään toimivana ratkaisuna.

Järjestelmän palvelinkokonaisuus tullaan toteuttamaan C# ohjelmointikielellä. Palvelin tulee olemaan hyvin perinteinen sekä melko yksinkertainen CRUD-operaatioita sisältävä REST-rajapinta, joka toimii pääasiallisesti välikätenä frontendin sekä tietokannan välillä.

Tietokanta toteutetaan hyödyntäen MongoDB:tä. MongoDB on NoSQL-tietokanta eli tietokanta, jossa ei ole relaatioita. Tämän tietokannan data koostuu objekteista, joilla voi olla täysin yksilöllisiä datarakenteita muihin tietokannan objekteihin verrattuna. Päätin käyttää tätä tietokantaa paitsi oman kiinnostukseni takia, mutta myös sen nopeuden sekä helppokäyttöisyyden takia. Tämä tietokanta on huomattavasti nopeampi verrattuna relaatiotietokantaan. Mikäli tietorakenne olisi monimutkaisempi, relaatiotietokanta olisi todennäköisesti järkevämpi ratkaisu.

Muuta

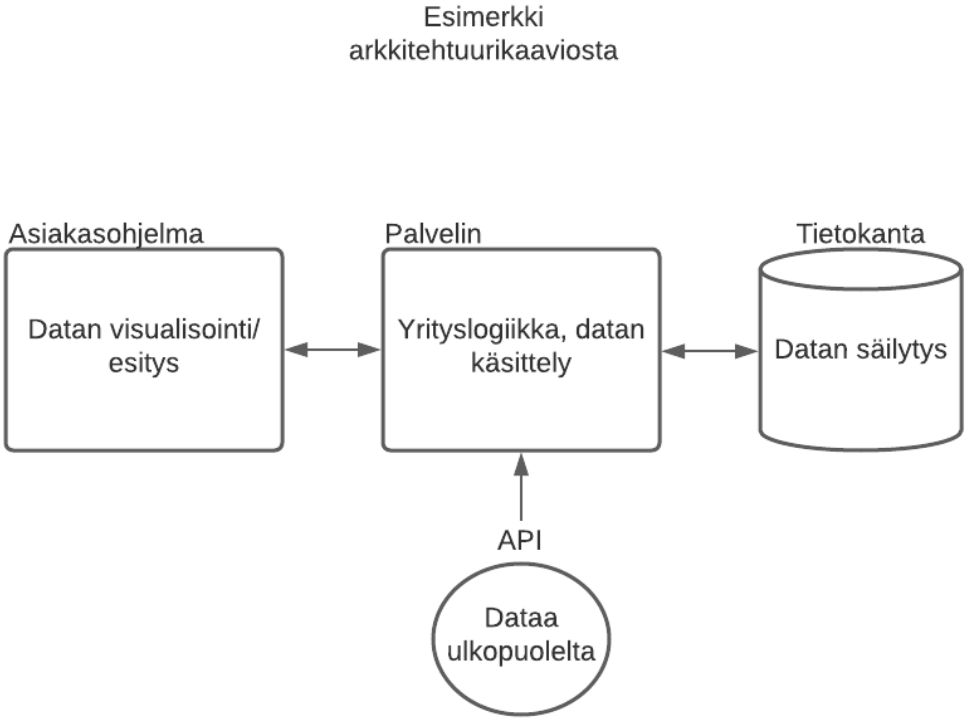
Projekti on kokonaisuutena melko laaja ja verkkopalvelun kehittäminen on itsessään jo melko työläs prosessi. Tämän lisäksi täytyy ottaa huomioon suunnitteluvaiheen kaikki osa-alueet ennen verkkopalvelun konkreettista toteutusta. Tästä syystä projektin sekä verkkopalvelun täydellinen viimeistely ei ole edellytyksenä tutkimuksen toteutumiselle, vaan toimiva prototyyppi riittää tähän käyttötarkoitukseen.

Tutkimusmenetelminä tälle tutkimukselle toimivat niin kirjallisuus, nettijulkaisut ja artikkelit, kuin itse tutkimuksen ohella toteutettava projekti. Projektin toteutus on kirjoitettu synkronoidusti projektin kanssa, kirjoittaen samaan tahtiin projektin etenemisen kanssa. Koin sen mielenkiintoiseksi ratkaisuksi, sillä lukija pystyy näkemään omat ajatukseni projektin aikana vaihe vaiheelta ja pääsee ratkaisuihini johtaneisiin päätelmiin sisään.

1 Ohjelmistoarkkitehtuuri

Alkuun on hyvä selvittää, mitä ohjelmistoarkkitehtuuri oikeastaan edes on. Tämä kysymys jakaa vielä tänäkin päivänä vahvasti mielipiteitä. Melko epämääräinen, mutta kuitenkin järkevä luonnehdinta tähän on, että arkkitehtuurissa on kyse tärkeistä asioista, mitä ne ikinä sitten ovatkaan (Fowler, 2019). Tämähän ei itsessään oikeastaan kerro vielä yhtään mitään, mutta samaisessa Fowlerin artikkelissa tätä kuvausta avataan lisää. Kyse on nimittäin arkkitehtuurillisesta ajattelutavasta (Fowler, 2019). Kuvauksella yritetään siis tuoda esiin sitä, että arkkitehtuurillisen ajattelun ytimessä olisi nimenomaan kyky päättää mitkä seikat ovat arkkitehtuurillisesti merkittäviä. Nämä arkkitehtuurilliset seikat ovat projekteissa täysin yksilöllistä.

Mielestäni sen lisäksi, että on tärkeää tietää mitä kehittää, on myös tärkeää tietää *miten* sen kehittää. Sitä miettiessä ohjelmistoarkkitehtuurin merkitys kasvaa ja juuri tästä syystä itse olen sitä mieltä, että ohjelmistoarkkitehtuuri on ohjelmiston kannalta *elintärkeää*.



1.1 Arkkitehtuurin tärkeys ohjelmistokehityksessä

Olen vahvasti sitä mieltä, että laadukkaalla ohjelmistoarkkitehtuurilla on suora relaatio ohjelmiston elinikään. Sovelluksia, joissa on hyvin implementoitu ohjelmistoarkkitehtuuri, ja joissa arkkitehtuuri on suunnittelun keskiössä, on helppo alkaa laajentamaan. Tämä siitä syystä, että sovellusta kehittäessä laajentamisen mahdollisuutta on osattu ennakoida

ja täten osattu myös ottaa osaksi arkkitehtuurillisia ratkaisuja. Jos ohjelmistoa ei voida ylläpitää, saati kehittää huonon arkkitehtuurin vuoksi, ohjelmisto rappeutuu ja muuttuu ajan myötä käyttökelvottomaksi.

Ilman minkäänlaista arkkitehtuurillista suunnittelua tai silmää ennen ohjelmiston varsinaista implementaatiota tulee kehittämisessä, etenkin jatkokehityksessä vastaan paljon haasteita. Sovellusjärjestelmän sekä sen komponenttien määrän kasvaessa sekä kokonaisuuden monimutkaistuessa myös ohjelmistoarkkitehtuurin merkitys kasvaa. Se tuo esiin aivan uudenlaisia ongelmia, joita ilman arkkitehtuuria olisi hyvin haasteellista tunnistaa. (Garlan & Shaw, 1994.)

Itse lisäisin tähän vielä, että arkkitehtuurin lisäksi tarvitaan myös siihen erikoistuneita ammattilaisia tunnistamaan alueeseen liittyviä ongelmia. Ohjelmistokehityksen näkökulmasta tarkasteltuna arkkitehtuuri on oma laajempi osa-alueensa. Olisi siis ainoastaan järkevää, että siihen liittyviin tehtäviin on määrätty siihen erikoistunut ammattilainen.

Tänä päivänä kuitenkin ohjelmistoarkkitehti on melko keskeinen sekä itsestään selvä projektin ”elin”. Itsestäänselvyydellä tietysti tarkoitan, että on lähes standardikäytäntö, että jokaisessa ohjelmistoprojektissa on tätä nykyä oma designoitu arkkitehti vastaamassa ohjelmiston rakenteellisuuden suunnittelusta sekä implementaatiosta. Arkkitehti vastaa yleensä jostain tietystä ohjelmiston komponentista tai projektin osasta (Salomäki, 2020). Tämän uskoisin kuitenkin täysin riippuvan projektin koosta sekä sen luonteesta, sillä pienellä projektilla voisi varsin hyvin olla vain yksi arkkitehti, joka vastaisi kaiken koodin rakenteesta. Toisaalta, jos projekti on laaja ja omaa monta eri osa-aluetta, lienee järkevämpää, että projektissa olisi mahdollisesti useampi arkkitehti.

Sen lisäksi, että ohjelmistoarkkitehti suunnittelee ohjelman varsinaisen arkkitehtuurin, tekee hän myös paljon muuta. Projektin jäsenten kanssa kommunikointi, kaavioiden sekä dokumentaatioiden laatiminen on varsin arkipäiväistä työtä ohjelmistoarkkitehdille. Lisäksi arkkitehtuurikatselmoinnit, koodin katselmoinnit, tiimin jäsenten mentorointi sekä riskiarviointi sekä -hallinta ovat tyypillisiä ohjelmistoarkkitehdin työkuvaan kuuluvia osa-alueita. (Salomäki, 2020.)

Periaatteessa kuitenkin nämä eritellyt pienemmät vastuut arkkitehdille, kuten juuri esimerkiksi edellä mainitut koodikatselmoinnit ja mentorointi ovat käytännössä kirjoittamaton sääntö alalla. Tarkoitan tässä sitä, että kokeneille kehittäjille on selvää, että kokeneemmat yleisesti katsottuna auttavat ja myös jollakin tasolla ”mentoroivat” junioreja.

1.2 Arkkitehtuurimallit

Arkkitehtuurilliset mallit ovat ikään kuin pohjapiirustuksia erilaisille järjestelmille. Tätä mallia noudattaen toteutetaan ja kehitetään jokin siihen sopiva sovelluskokonaisuus. Jokainen yksittäinen malli on suunniteltu toteuttamaan tietynlaisia ohjelmisto- tai sovelluskokonaisuuksia. Esimerkiksi verkkokauppoja kehittäessä saatetaan hyödyntää toista, ja mobiilisovellusta kehittäessä taas aivan eri arkkitehtuurimallia.

Tunnetuimpia ohjelmistoarkkitehtuurimalleja lienee MVC-malli. MVC-mallin (*Model-View-Controller*) periaate on erottaa käyttöliittymä muusta ohjelmistologiikasta. Lyhykäisyydessään arkkitehtuurin ajatus on se, että Controller -osio on varsinainen toimija. Tämä kysyy Modelilta esimerkiksi dataa. Sen jälkeen Controller välittää halutun datan Viewille eli käyttöliittymälle. Näin saadaan ohjelma purettua pienemmiksi omiksi järkeviksi kokonaisuuksiksi. Näitä pienempiä kokonaisuuksia on sitten helpompi lähteä korjaamaan tai tarvittaessa jatkokehittämään ilman, että sillä olisi rakenteellista vaikutusta sovelluksen muihin osiin tai moduuleihin.

1.2.1 Kolmen tason arkkitehtuuri

Tätä projektia varten päädyin kuitenkin hyödyntämään kolmen tason arkkitehtuurimallia (*Three-tier architecture*). Malliksi se on melko yksinkertainen, mutta todella tehokas ja siitä syystä myös yleinen. Tämä kyseinen arkkitehtuurimalli oli täydellinen ratkaisu projektille, sillä siinä on juuri tarvittavat osat ja kuten mainittu, se on suhteellisen yksinkertainen toteuttaa, mikä taas tutkimuksen näkökulmasta on suotavaa.

Kolmen tason arkkitehtuuri lajittelee sovelluksen kolmeen loogiseen sekä fyysiseen tasoon: esitystaso *(presentation tier)*, sovellustaso *(application tier)* sekä datataso *(data tier).* Esitystaso vastaa sovelluksen käyttöliittymää. Sovellustaso tyypillisesti pitää sisällään kaiken yrityslogiikan sekä toimii yleisellä tasolla esitystason ja datatason viestinviejänä. Datataso toimii sitten sovelluksen datan tallennuspaikkana eli pitää sisällään tietokannan, jonne ohjelmistodata talletetaan. (IBM Cloud Education, 2020.)

Kolmen tason arkkitehtuurin suurin hyöty piilee siinä, että koska jokainen taso pyörii periaatteessa omassa infrastruktuurissaan, pystyy jokaista tasoa kehittämään vaikuttamatta muihin arkkitehtuurin tasoihin. Jokaiselle tasolle voisi myös esimerkiksi varata oman tiimin projektissa, mikä yleensä onkin käytäntönä. (IBM Cloud Education, 2020.)

Ohjelmiston hajauttaminen on kauttaaltaan ohjelmoinnissa ideanakin. Eli siis tavoitteena saada koodi koostumaan pienistä toisistaan riippumattomista komponenteista, jotta

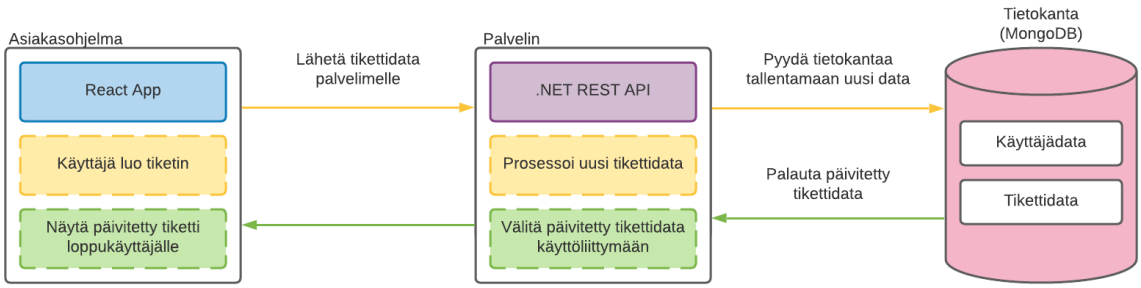
jokaisen komponentin mahdollinen jatkokehittäminen, korjaus, päivitys sekä potentiaalinen uudelleenkäyttö olisi mahdollisimman suoraviivaista. Tälle on myös englanninkielinen nimitys, *loose coupling*.

Muita kolmen tason arkkitehtuurille ominaisia hyötyjä ovat nopea ohjelmistokehitys, skaalautuvuus, luotettavuus sekä tietoturva. Nopeaan ohjelmistokehitykseen vaikuttaa se seikka, että jokaista arkkitehtuurin tasoa voi kehittää yhtäaikaisesti, sillä ne eivät ole riippuvaisia toisistaan. Skaalautuvuuteen vaikuttaa myös tämä seikka, sillä jokaista tasoa voi yksin laajentaa halutessaan ilman, että se vaikuttaa muhin tasoihin. Luotettavuus tulee esille esimerkiksi siinä, että jos jokin edellä mainituista tasoista esimerkiksi kaatuisi, ei se vaikuttaisi muhin tasoihin lainkaan. Tämä tietysti täytyy ottaa järjestelmän infrastruktuurissa huomioon, esimerkiksi varaservereillä, joissa on varmuuskopioita tietokannasta. Viimeisimpänä tietoturva. Koska esitystaso ja datataso eivät voi suoranaisesti keskustella toistensa kanssa, hyvin suunniteltu sovellustaso voi toimia ikään kuin sisäisenä palomuurina, joka estää esimerkiksi SQL injektiot ja muut haitalliset toimenpiteet. (IBM Cloud Education, 2020.)

2 Projektin toteutus

2.1 Pohjustus

Alkuun oli tärkeää kartoittaa projektin laajuus sekä suunnitella sovelluksen arkkitehtuuri kaavion muodossa. Ajatus oli, että ensin toteutetaan niin koko sovelluksen arkkitehtuurikaavio kuin myös backendin sekä frontendin arkkitehtuurikaaviot. Tulin kuitenkin siihen tulokseen, että kukin arkkitehtuurikaavio oli järkevintä toteuttaa erikseen, kun työstän kyseistä projektin osa-aluetta. Siispä ensin toteutin koko järjestelmän kattavan, pelkistetyn kaavion. Kaavio esitettynä alempana.



liite 1

Tässä pohjustusvaiheessa en kuitenkaan vielä suunnitellut itse algoritmia, joka toimii ikään kuin koko toiminnallisuuden keskiössä. Tämä algoritmi tulee olemaan se, joka laskelmoi sekä delegoi tiketit sopiville projektin jäsenille. Vielä tässä pohjustusvaiheessa ei ollut tiedossa myöskään sitä, minne algoritmin rakentaisin. Toisaalta sen voisi hyvin rakentaa frontendiin, mutta kuulostaisi järkevämmältä, että sen kaltainen logiikka toimisi palvelimella. Tämän algoritmin kaltainen monimutkaisempi ohjelmistologiikka tyypillisesti toimii muun yrityslogiikan kanssa palvelimella.

Tähän projektin vaiheeseen sisältyi tietysti myös kehitysympäristöjen alustaminen. Eli tässä vaiheessa tietokanta, backend -sovellus sekä frontend -sovellus ovat kaikki ympäristöjensä puolesta valmiina.

2.2 Backend sekä tietokanta

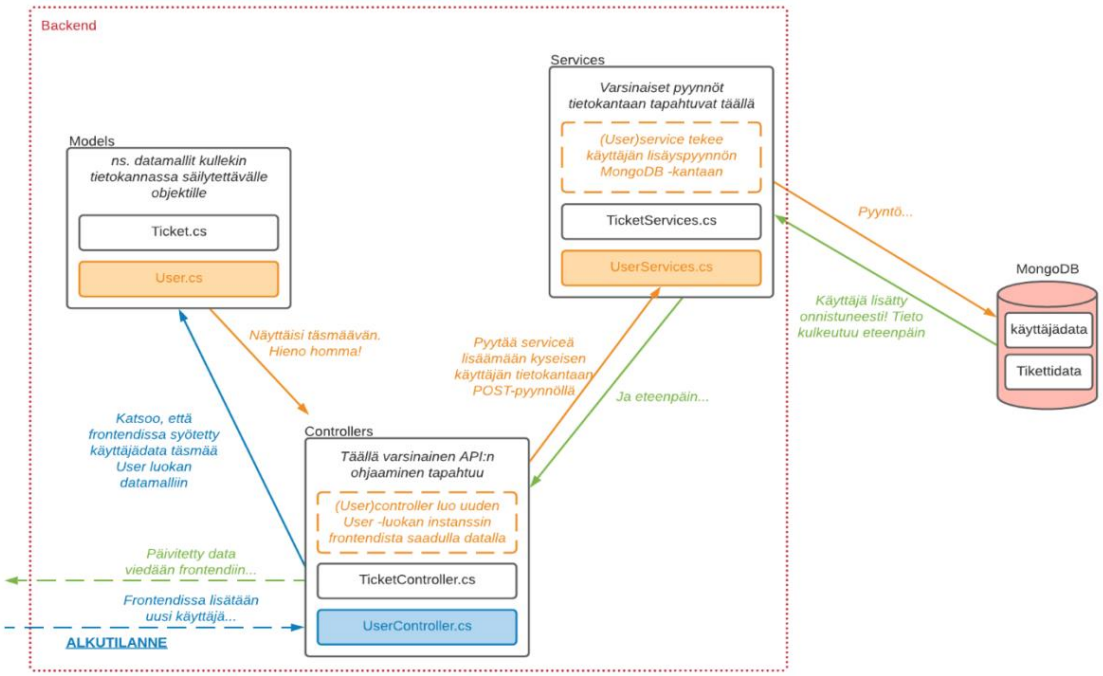
Projektin niin kutsutussa toisessa vaiheessa toteutetaan backend, sekä myös tietokanta. Backendin eli sovelluksen palvelintason tehtävä on keskittyä sovelluksen toiminnallisuuksiin. Päivitykset sekä muutokset sovelluksessa ovat ensisijaisesti palvelimen vastuulla. Palvelimelle kirjoitettu koodi kommunikoi tietokannassa sijaitsevan tiedon sovelluksen käyttöliittymään loppukäyttäjän nähtäville. (Stewart, 2021.)

Tähän vaiheeseen sisältyy myös tietokannan objektien suunnittelu sekä muodostus. Lisäksi backendistä muodostetaan yhteys tietokantaan ja varmistetaan, että käyttäjiä sekä tikettejä tosiaan pystyy tallentamaan, poistamaan, hakemaan, lukemaan sekä päivittämään tietokantaan. Lopuksi backend julkaistaan Herokuun, joka on julkinen pilvipalvelualusta, jonka tarkoitus on mahdollistaa asiakkaille omien sovellusten julkaisu internetiin muiden käytettäväksi.

Tämän projektin sovelluksen palvelin toimii REST-rajapintana tietokannan sekä frontendin välistä keskustelua varten. Tämänkaltaisiin CRUD-operaatioita tekeviin rajapintasovelluksiin löytyy erittäin paljon materiaalia internetistä, joten sen tekemiseen tulisi löytyä tarvittaessa runsaasti apuja.

Toteutus oli itsessään melko suoraviivainen. Kuitenkin backend -kehityksen ollessa itselle edelleen hieman vierasta, jouduin turvautumaan opetusmateriaaliin, jossa esitettiin kuinka REST API .NET:iä hyödyntäen oikein rakennetaan. Lopputuloksena kaikesta huolimatta on hieman MVCS-arkkitehtuuria muistuttava palvelinsovellus.

MVCS-arkkitehtuurimalli polveutuu melko suuressa suosiossa olevasta MVC-mallista, jonka pääperiaatteena on eristää palvelimen toiminnallisuudet kolmeen eri osioon: Model, View, sekä Controller (Kumar, 2021). MVCS-mallissa mukaan kuitenkin liitetään vielä niin kutsuttu *service layer*, jota tämän projektin tapauksessa hyödynnetään keskusteluun tietokannan kanssa. Palvelinsovelluksen arkkitehtuurikaavio esitettynä alempana.



liite 3

Palvelimen valmistuessa ja toimiessa lokaalisti omalla tietokoneellani ajattelin, että projektin palvelinosio olisi ollut siinä. Näin ei kuitenkaan ollut, sillä yrittäessäni julkaista palvelinsovellusta Herokun omille palvelimille julkiseen käyttöön huomasinkin pian, ettei palvelimen julkaisu Herokuun ole mahdollista, sillä kyseinen alusta ei tue .NET-sovelluksia natiivisti. Pienen internet-tutkiskelun jälkeen ilmenikin, että ongelman pystyy kiertämään laittamalla itse sovellus eli palvelin toimimaan konttiin.

Kontit ovat siis ohjelmistopaketteja, jotka pitävät sisällään ohjelman suorittamiseen vaadittavat tiedostot sekä riippuvuudet, jotta kontin sisältämää sovellusta voidaan ajaa. Ja koska ohjelma on sisällytetty konttiin, voidaan sitä ajaa mistä tahansa isäntäjärjestelmästä. (Pittet, 2021.) Konttien ajoon käytin äärimmäisen suosittua sekä tunnettua Dockerin tarjoamaa alustaa konttien tekemiseen sekä ajamiseen. Docker on alustapalvelu konttien luontia sekä ajoa varten (Docker overview: Docker, 2021).

Palvelinsovelluksen asentaminen konttiin oli todella suuri haaste itselle, sillä aikaisempaa kokemusta konteista ei ole ollut. Tämän lisäksi myös kontin saaminen Herokuun oli haastavaa, mutta siinäkin loppujen lopuksi onnistuin. Kuva palvelinsovelluksesta toiminnassa Herokussa esitettynä alempana.

liite 4

2.3 Frontend

Seuraavaksi viimeiseen projektin vaiheeseen eli frontendiin. Tyypillisesti frontendin tehtävä on hallinnoida kaikkea mitä loppukäyttäjä näkee tai painaa käyttöliittymässä selaimessa tai sovelluksessa. Frontend -kehittäjät ovat pääsääntöisesti vastuussa sovelluksen visuaalisesta esityksestä sekä sovelluksen kokonaisvaltaisesta käyttäjäkokemuksesta. (Stewart, 2021.)

Sovelluksen frontend oli ehdottomasti projektin työläimpiä kokonaisuuksia ja tämä ilmenee esimerkiksi siitä, että sen toteutus vaati noin 800 riviä koodia, mikä on noin kolminkertainen määrä esimerkiksi projektin palvelimeen verrattuna. Projektin toteutuksesta jäi kuitenkin valitettavasti uupumaan yksi sen uniikeimpia piirteitä, nimittäin itse jakelualgoritmi. Onneksi tämä ei kuitenkaan tutkimuksen kannalta ollut kriittinen osa sovelluksen kokonaisuutta. Tässä osiossa pureudutaan syvemmin projektin frontendin arkkitehtuurillisiin ratkaisuihin.

Kuten jo johdannossa kävi ilmi, sovelluksen frontend toteutettiin hyödyntäen modernia JavaScript -kirjastoa nimeltään React. Tämä kirjasto on Facebookin kehittämä sekä vuonna 2013 julkaisema ilmainen sekä avoimeen lähdekoodiin perustuva kokonaisuus.

Kehittäessä React -sovellusta tärkeimpiä arkkitehtuurillisia piirteitä lienee uudelleenkäytettävät komponentit, joista käyttöliittymä rakennetaan. Jokaisella komponentilla on oma selkeä tavoitteensa ja yksi selkeä tarkoitus. Ideaalinen tilanne olisi, että yksittäinen komponentti on suunniteltu tekemään vain yhtä asiaa. (React, ei pvm.) Englannin kielessä tätä kutsutaan nimellä *single-responsibility principle.*

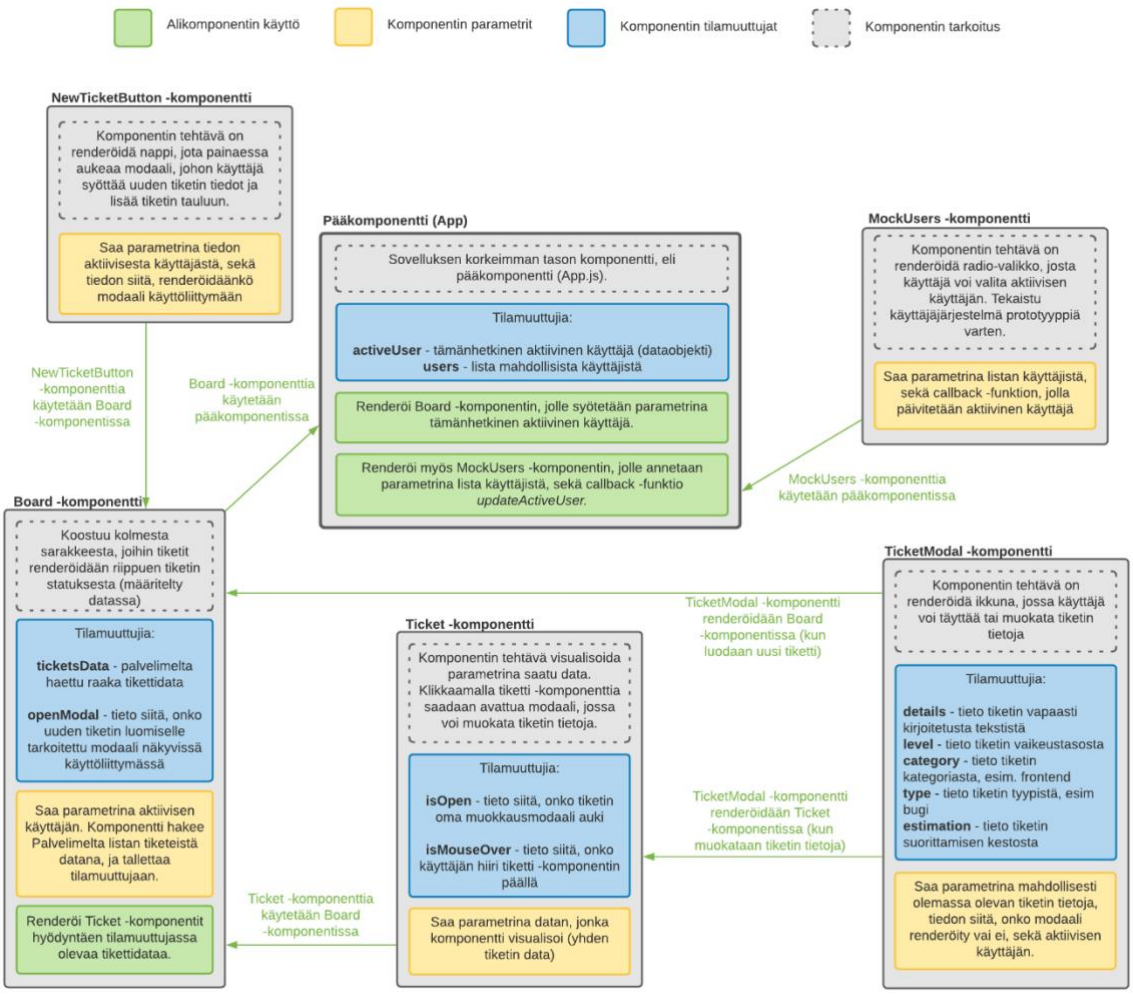
Eli on siis huonoa praktiikkaa, jos komponentti yrittää tehdä liian montaa asiaa. Hyvä sääntö tähän on, että jos komponentti tekee useampaa asiaa kerralla yhden sijaan, tulee se hajottaa pienemmiksi alikomponenteiksi. Koitin pitää sääntöä komponentteihin liittyen mielessä koko frontend-kehityksen ajan. Vaikka komponenttini olivat suurimmalta osin rakenteellisesti hyviä, ilmeni kokonaisuudessa kuitenkin pieniä ongelmia tilanhallinnan kanssa.

2.3.1 Tilanhallinta

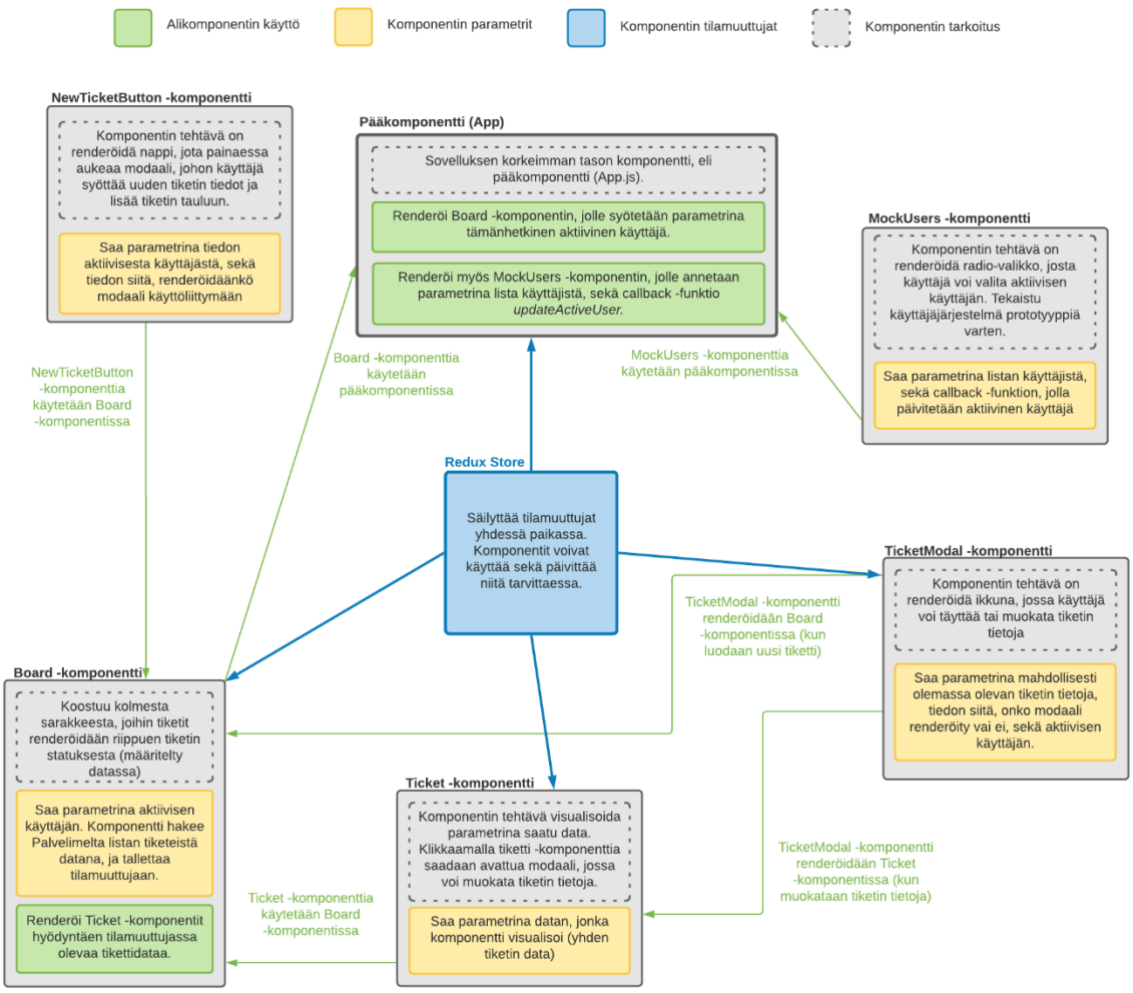
Lyhykäisyydessään ”tila” *(state)* tarkoittaa dataa, joka muuttuessaan päivittää myös käyttöliittymää. Tila tarkoittaa ohjelman tai ohjelman osan, eli esimerkiksi komponentin sen hetkistä tilaa (Schwarzmüller, 2021). Sovellusprojektin tapauksessa tilanhallinta ei

toteutunut toivotulla tavalla. Esimerkiksi uutta tikettiä lisätessä pitäisi käyttöliittymän päivittyä automaattisesti, jotta uusi tiketti näkyisi taulukossa. Tiketti näkyy kuitenkin vasta, kun käyttäjä itse päivittää selaimen ikkunan.

olen kuitenkin jo keksinyt tapoja ratkaista kyseisen tilanhallintaongelman. Yksi vaihtoehto on vain yksinkertaisesti selvittää, miksi komponentti ei päivity tilan päivittyessä, ja säilyttää nykyinen hajautettu tilanhallinta. Sen selvittäminen on jo entuudestaan tuttua eikä luultavasti veisi kauheasti aikaa. Toinen arkkitehtuurillisesti ehkä laadukkaampikin ratkaisu olisi käyttää Redux -nimistä tilanhallintakirjastoa. Kirjastolla saadaan kaikki tilamuuttujat sidottua fyysisesti yhteen paikkaan, ja komponentit voivat kyseisiä muuttujia tarvittaessa sekä käyttää että muuttaa. Tämänhetkinen ratkaisuni toimii niin, että tilamuuttujat ovat käytännössä hajautettuna loogisiin paikkoihin pitkin projektin lähdekoodia. Alempana havainnollistettu arkkitehtuuri ensin ilman Reduxia, ja sen jälkeen Redux -kirjastoa hyödyntäen.



Liite 2. Tila hajautettu komponentteihin.



Liite 2.5. Tila keskitetty yhteen paikkaan Redux -kirjastolla. Ratkaisu tilaongelmaan.

2.4 Keskeisimmät ongelmat kehityksen aikana

Jokaisella projektilla on omat ongelmansa. Ongelmat sekä virheet ovat tärkeä osa projekteja, sillä niistä oppii aina jotakin mitä voi tulevaisuuden projekteissa hyödyntää. Tutkimuksen ohella toteutetussa projektissakaan en välttynyt ongelmilta. Niitä oli jokseenkin paljon eli paljon myös opin uutta.

2.4.1 Palvelimen ongelmat

Palvelinta kehittäessä suurin ongelma oli ehdottomasti yhteensopivuus dotNET -sovelluksen sekä Herokun välillä. Heroku ei siis toisin sanoen tue C# kielellä kirjoitettuja sovelluksia natiivisti. Tämän kiertämiseksi jouduin turvautua täysin itselle vieraisiin teknologioihin yrittäessäni konfiguroida Docker -konttia, johon sovellus tuli asentaa toimimaan. Tuo oli itselle erittäin haasteellinen toteuttaa, että kun vihdoin sain kehittämäni REST-rajapinnan Herokuun, en enää opinnäytetyötä varten suostunut muutoksia sen koodiin tekemään. Koin, että korjausten tekemiseen kuluisi aivan liikaa aikaa eikä itse projekti ollut *niin* kriittinen tutkimuksen toteutumiselle.

Myöhemmin projektin aikana kuitenkin vielä selvisi, että palvelimen eli sovelluksen rajapinnan tieto oli puutteellista algoritmin toteutumisen näkökulmasta. Esimerkiksi melko keskeinen data puuttuu kokonaan, nimittäin tieto siitä kenelle tiketti on määrätty. Ilman tätä tietoa on oikeastaan mahdotonta tallettaa tietokantaan tieto henkilöstä, joka on vastuussa tiketin toteutumisesta. Tuon muutoksen lisääminen palvelimen koodiin on ajatustasolla melko suoraviivainen juttu. En kuitenkaan ole riittävän kokenut Dockerin käyttäjä, joten en halunnut ottaa riskiä palvelimen toimivuuden suhteen. *Alla havainnollistus puutteellisesta datasta rajapinnassa.*


```
{
  "id": "60eaf776905358b9ce1efdd3",
  "creationDate": "2021-06-25T09:51:15.575Z",
  "lastModified": "2021-06-25T09:51:15.575Z",
  "creator": {
    "id": null,
    "firstName": "etunimi",
    "lastName": "sukunimi",
    "email": "etunimi.sukunimi@mail.com",
    "password": "salasana",
    "title": "titteli",
    "level": "taso",
    "activeTickets": [
      null
    ]
  },
  "textContent": "tiketin tekstisisältö tulee tähän",
  "level": "tiketin vaatimustaso",
  "category": "tiketin kategoria",
  "type": "tiketin tyyppi",
  "status": "new",
  "estimationInHours": 7,
  "comments": [

  ]
}
```


Myös yleisesti backend -kehitys oli itsessään hieman haasteellista kokemattomuuteni vuoksi ja aiheutti ajoittain ongelmia. Yksinkertaisten CRUD -operaatioita tekevien REST-rajapintojen tekeminen on kuitenkin niin yleistä tänä päivänä, että mille tahansa backend -kielelle löytyy varmasti ohjeistusta internetistä.

2.4.2 Asiakasohjelman ongelmat

Asiakasohjelman suurimpia ongelmia käytiin jo hieman läpi kohdassa *2.3.1 Tilanhallinta,* joten tässä luvussa ei enää ole tarpeellista sitä suuremmin avata. Kuitenkin toinen, ehkä

käyttäjän näkökulmasta keskeinen ongelma, on ongelma sovelluksen responsiivisuuden kanssa. Toistaiseksi sovellus on suunniteltu käytettäväksi vain tietokoneen näytöllä. Mitään muuta mahdollisuutta ei juuri ole.

2.5 Jatkokehitys

Jatkokehitys on ollut koko ajan osana projektisuunnitelmaa. Heti projektin alussa oli selvää, että projektin aikaansaama sovellus tulee olemaan erittäin alkeellinen sekä yksinkertainen, ja että sitä tulisi hioa tulevaisuudessa. Mahdollisuuksia jatkokehitykselle on monia. Uusia ominaisuuksia, kuten myös yleistä koodin optimointia on hyvin mahdollista toteuttaa. Aikomukseni olikin koko projektia suunnitellessa saada riittävä prototyyppi aikaiseksi, jota tulevaisuudessa voi parannella sekä laajentaa.

Ensisijainen ajatus on muuttaa frontend-koodia niin, että sitä on helpompi skaalata ja jatkokehittää. Esimerkiksi tilanhallinta *(siitä lisää osiossa ”2.3.1 Tilanhallinta”)* lienee asia, joka tulisi ratkaista pikimmiten. Lisäksi palvelimessa toimivaan REST-rajapintaan tulisi tehdä datan puolesta muutoksia, jotta itse algoritmi saataisiin järkeväksi kokonaisuudeksi. Kun data on saatu korjattua, voidaan siirtyä implementoimaan edellä mainittua algoritmia. Sovellukseen olisi hyvä myös implementoida kunnollinen kirjautumisjärjestelmä. Sovelluksen ensinäkymä voisikin olla sisäänkirjautumisruutu.

Käyttäjän näkökulmasta yksi parannusehdotus voisi olla sovelluksen responsiivisuus. Tällä hetkellä sovellus on tarkoitettu käytettäväksi vain tietokoneen näytöllä. Tätä voisi parantaa esimerkiksi mahdollistamalla erinomainen käyttäjäkokemus myös mobiililaitteilla.

Mahdollisuuksia on vielä enemmän kuin mitä edellä mainittiin. On kuitenkin tärkeää priorisoida jatkokehitysideat, ja ylipäätään miettiä ratkaisuja, jotka parantavat sovellusta esimerkiksi käyttäjän näkökulmasta tai sovelluksen elinikää parantavasti.

3 Yhteenveto sekä pohdinta

Projekti on saatu melko tavoitteellisesti suoritettua loppuun ja tutkimustuloksiakin on syntynt. Näin tutkimusdokumentin loppupuolella on hyvä aika kummankin tutkimustuloksille, yhteenvedolle sekä jälkipohdinnalle.

3.1 Tulokset

Opinnäytetyön tavoitteena oli selvittää, mitä hyvä ohjelmistoarkkitehtuuri ohjelmistokehityksessä tarkoittaa ja toteuttaa toimiva prototyyppi automatisoidusta tikettitaulu-sovelluksesta. Tutkimuksessa selvisi mitä käytänteitä ja mitä kehittäjän tulisi pitää mielessä, jotta hyvän ohjelmistoarkkitehtuurin piirteet voidaan saavuttaa. Projektin osalta ei täysin haluttuun lopputulemaan päästy, sillä projektin kannalta ehkä keskeisin osa, eli automaatio-ominaisuuden tuova jakelualgoritmi, jäi toteuttamatta. Onneksi algoritmi ei kuitenkaan tutkimuksen kannalta ollut kriittinen.

Mitä projektin sovelluksen arkkitehtuurillisiin ratkaisuihin tulee, ei kaikki aivan täydellisesti onnistunut. Palvelimen arkkitehtuurissa itsessään ei ole erityisesti vikaa. Palvelin rakenteeltaan on onnistuttu jakamaan pienempiin toimiviin kokonaisuuksiin, joita on helppo muokata sekä laajentaa tarvittaessa. Asiakasohjelmassa on arkkitehtuurin puolesta kuitenkin vielä tilaa parannuksille. Tilanhallinta lienee akuutein asia korjattavaksi. Tilamuuttujia voisi esimerkiksi hallita Redux -kirjastolla, joka sovellukseen sopisi erinomaisesti.

3.2 Pohdinta

Ohjelmistoarkkitehtuurin tutkiminen oli erittäin mielenkiintoista. Ohjelmistoarkkitehtuuri jo *itsessään* on mielenkiintoista. Harva pysähtyy ajattelemaan, että huonosti toteutettu koodi sekä arkkitehtuuri on suoraan yhteydessä sovelluksen elinikään. Jossain kohtaa huonosti toteutettua sovellusta ei ole enää järkevä jatkokehittää koodin ollessa liian epäselvää ja huonosti organisoitua. Huonosti toteutettu arkkitehtuuri voi olla jopa liiketoiminnallisesti vaarallista.

3.3 Henkilökohtainen kehitys

Kuluneen projektin sekä tutkimuksen aikana opin huimasti uutta. Asiakasohjelmia jatkossa koodatessani osaan kiinnittää huomiota tilanhallinnallisiin ratkaisuihin. Opin myös vaikeimman kautta sen, että ennen toteutusta on hyvä selvittää työkalujen yhteensopivuuden. Viittaan tässä erityisesti Herokun sekä .NET -sovellusten yhteensopimattomuuteen.

Mitä ohjelmistoarkkitehtuuriin tulee, ei aikaisempaa perehtymistä juuri ollut. Opin uusia tunnettuja arkkitehtuurillisia malleja, jotka ovat erittäin laajasti omaksuttuja ihan maailmanlaajuisella tasolla. Ennen myös ihmettelin miksi järjestelmissä on aina palvelin asiakasohjelman sekä tietokannan ”välissä”. Palvelimella on kuitenkin suuri vastuu esimerkiksi sovelluksen turvallisuuden näkökulmasta. Hyvin toteutettu palvelin mahdollistaa sen, ettei asiattomia ja vaarallisia toimenpiteitä voida järjestelmässä tehdä.

Kuitenkin tärkein oppimani asia ohjelmistoarkkitehtuurista on se, että arkkitehtuuri on suoraan yhteydessä sovelluksen elinikään. Huonosti toteutettu arkkitehtuuri voi pahimmassa tapauksessa johtaa liiketoiminnan romahtamiseen.

Lähteet

Docker overview: Docker. (2021). Noudettu osoitteesta https://docs.docker.com/get-started/overview/. Viitattu Elokuussa 2021.

Fowler, M. (1. Elokuu 2019). *martinfowler.com*. Noudettu osoitteesta https://martinfowler.com/architecture/. Viitattu Elokuussa 2021.

Garlan, D.;& Shaw, M. (Tammikuu 1994). *An Introduction to Software Architecture.* Pittsburgh: Carnegie Mellon University. Noudettu osoitteesta https://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf. Viitattu Elokuussa 2021.

IBM Cloud Education. (28. Lokakuu 2020). *Three-Tier Architecture*. Noudettu osoitteesta IBM: https://www.ibm.com/cloud/learn/three-tier-architecture. Viitattu Marraskuussa 2021.

Kumar, A. (30. Maaliskuu 2021). *An Introduction To MVCS Architecture: Quantiphi*. Noudettu osoitteesta https://quantiphi.com/an-introduction-to-mvcs-architecture/. Viitattu Elokuussa 2021.

Pittet, S. (2021). *What is a Container?: Atlassian*. Noudettu osoitteesta https://www.atlassian.com/continuous-delivery/microservices/containers. Viitattu Elokuussa 2021.

React. (ei pvm). *Thinking in React*. Noudettu osoitteesta reactjs.org: https://reactjs.org/docs/thinking-in-react.html. Viitattu Marraskuussa 2021.

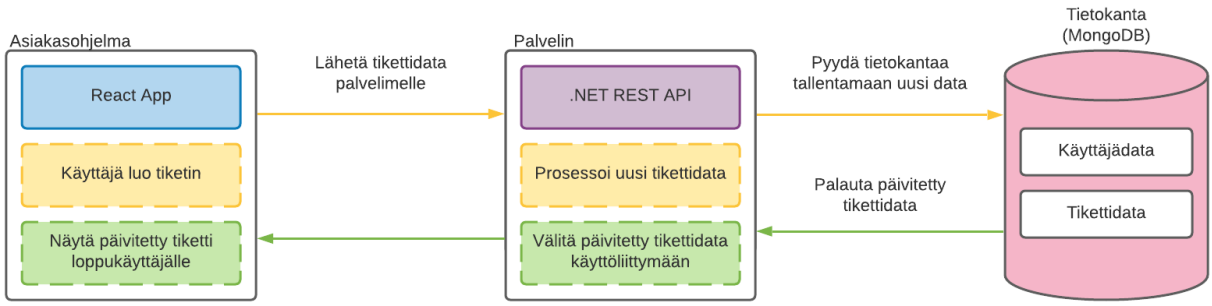
Salomäki, T. (10. Helmikuu 2020). *ITarkkitehti*. Noudettu osoitteesta https://itarkkitehti.fi/roolijako-vastuualueet-it-arkkitehtuurissa/. Viitattu Elokuussa 2021.

Schwarzmüller, M. (20. Tammikuu 2021). *What is State in Programming?* Noudettu osoitteesta Academind: https://academind.com/tutorials/what-is-state. Viitattu Marraskuussa 2021.

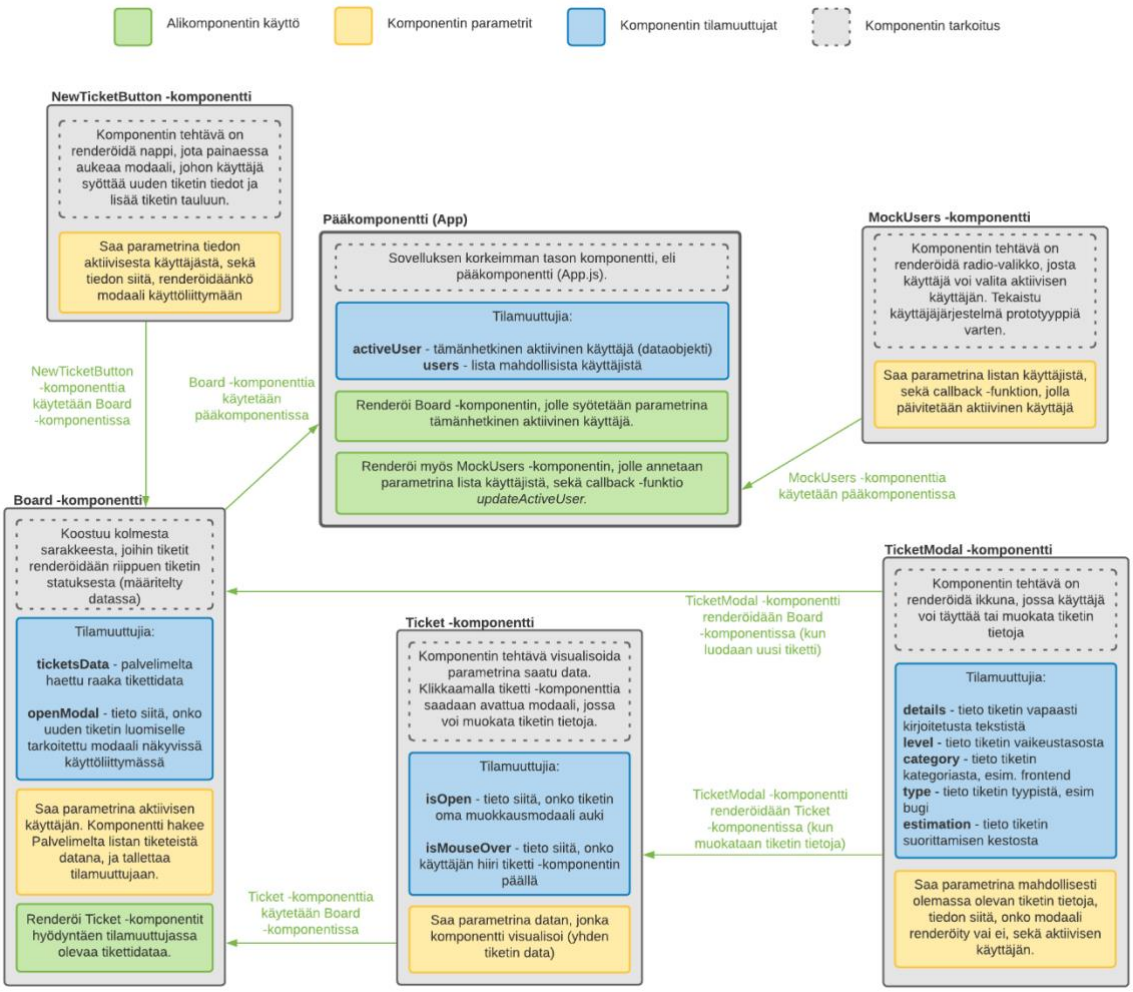
Stewart, L. (22. July 2021). *Front End Development vs Back End Development: Where to start?* Noudettu osoitteesta Course Report: https://www.coursereport.com/blog/front-end-development-vs-back-end-development-where-to-start. Viitattu Marraskuussa 2021.

Liitteet

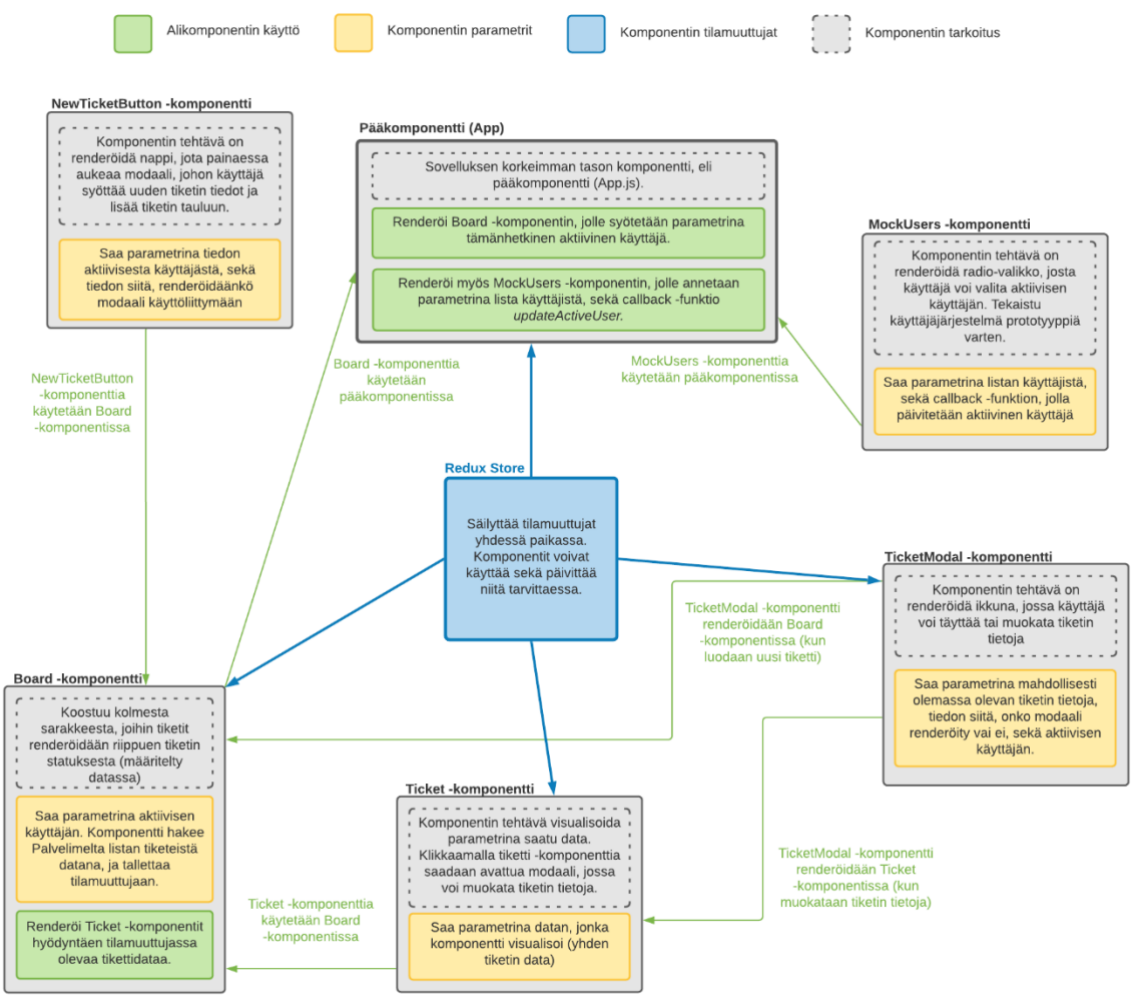
Liite 1. Järjestelmän arkkitehtuuri



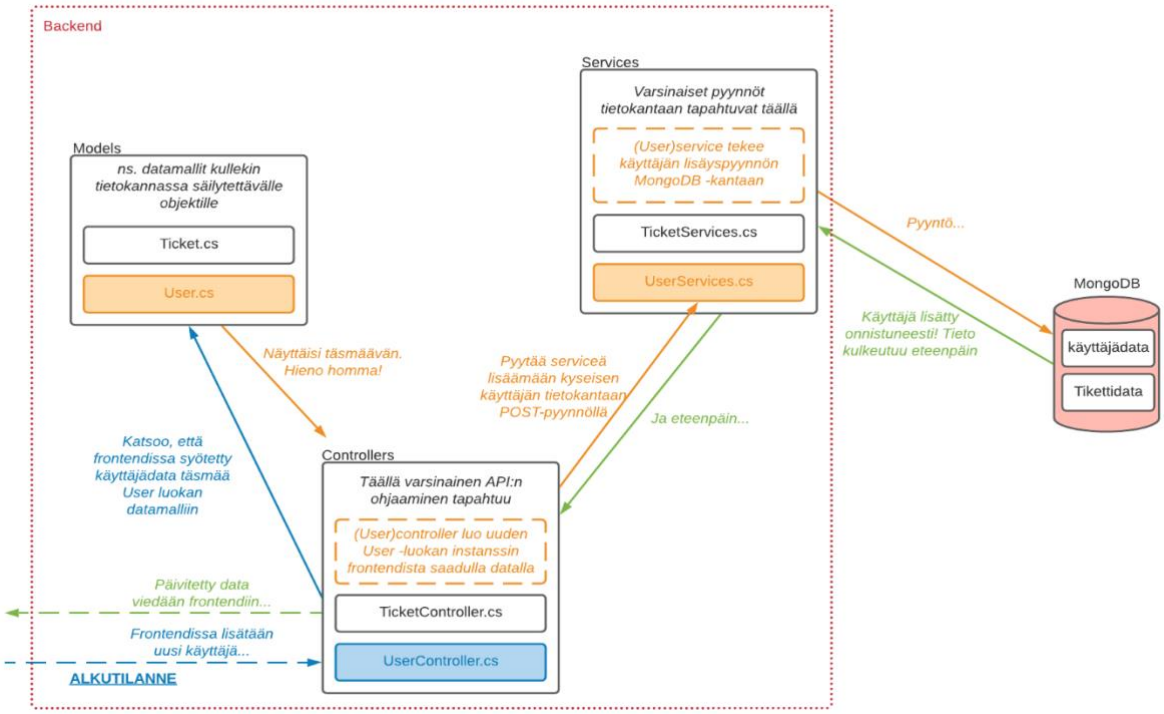
Liite 2. Frontendin arkkitehtuuri



Liite 2.5. Frontendin arkkitehtuuri Reduxilla



Liite 3. Backendin arkkitehtuuri



Liite 4. API:n julkaisu Herokuun