Tero Sorvisto & Aleksi Skantz

B2B-markkinointijärjestelmän toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

22.5.2017

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Aleksi Skantz & Tero Sorvisto B2B-markkinointijärjestelmän toteutus 51 sivua + 1 liitettä 22.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander

Tässä insinöörityössä esittellimme tavat ja tekniikat, joiden avulla suunnittellimme ja kehitimme asiakkaalle B2B-verkkokaupan ja sen ympäristön MVP-toteutuksen.

Projektin aikataulu oli suhteellisen lyhyt, joten päädyimme rajaamaan ominaisuuksia mahdollisimman paljon niin, että pystyisimme keskittymään olennaisimpiin toiminnallisuuksiin. Näin päädyimme toteuttamaan MVP-tuotteen, eli pienimmän mahdollisen toteutuksen tuotteesta, jota voidaan testata asiakkailla.

Määrittelyyn kuului verkkosovelluksen aloitussivu, tuotteiden rajaus kategorioiden mukaan, tuotesivut ja tilausten jättäminen.

Saimme asiakkaalta vapaat kädet valita sovelluksen kehittämiseen käytettävät tekniikat. Ainoa asiakkaan vaatimus oli, että me käyttäisimme Azure-pilvipalvelua kehitys ja tuotanto -alustana.

Ohjelmointiparadigmaksi valitsimme funktionaalisen ohjelmoinnin. Itse pääkieleksi valitsimme Clojuren. Clojure ei kuitenkaan sovelutunut front end –kehittämiseen, joten päätimme myös käyttää ClojureScript-kieltä.
ClojureScriptin tueksi valitsimme Reagent-kirjaston, eli kirjaston, joka toimi ClojureScriptin ja Reactin välisenä rajapintana. Tämä helpotti käyttöliittymän kehitystä.
Sivuston CSS:n luomiseen taas käytimme Clojurelle tehtyä Garden-kirjastoa.

Sovelluksen back end toteutettiin käyttäen serverless-arkkitehtuurimallia, jossa back endin toiminnallisuus suoritetaan käyttämällä Azuren funktioita.
Funktioiden ohjelmoimiseen käytimme Clojurescriptiä.
Tuotedataa säilytimme NoSql-tietokannassa Azuren DocumentDB:ssä.

Työssä käymme myös läpi jatkuvan integraation ja siihen liittyvät tekniikat.

Avainsanat	Azure, Clojure, ClojureScript, palvelimeton, pilvi, Reagent

Author(s) Title Number of Pages Date	Aleksi Skantz & Tero Sorvisto The Implementation of a B2B Marketing System 51 pages + 1 appendices 22 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer

This bachelor's thesis goes through the procedures and techniques used to design and develop a minimum viable product of a B2B-webshop.

The schedule for the project was relatively short so it was decided to cut down the features as much as possible to focus on the most relevant functionalities. For these reasons it was decided to make a minimum viable product, a minimum implementation of a viable product that could be tested with real customers.

The specifications for the web application included a starting page, to be able to filter products by choosing categories, product pages and being able to send an order for products.

The customer gave the freedom of choosing the technologies used for the project. The only requirement set by the customer was to use Azures cloud platform as a development and production platform.

Functional programming was chosen as the programming paradigm.
As the main programming language Clojure was chosen. It was soon discovered that Clojure was not the proper tool to use in the front end, so it was decided to also use ClojureScript.
To support ClojureScript the Reagent-library was chosen. It works as an interface between ClojureScript and React. This made the development of the user inter-face easier and faster. The CSS of the site was done using a library made for Clojure called Garden.

The back end was created using a serverless architecture model where the back end functionality is executed using Azures Functions service. The language used in the back end was also ClojureScript. The product data was stored in a NoSql database in Azures DocumentDB.

The study also intoduces the continuous integration and deployment as well as the related technologies.

Keywords	Azure, Clojure, ClojureScript, cloud, serverless, Reagent

Sisällys

Lyhenteet

MVP	Minimun Viable Product. Pienin mahdollinen toteutus tuotteesta, jota voidaan testata asiakkailla.
HTML	Hypertext Markup Language. Standardoitu kieli verkkosivujen ja verkko-ohjelmien tekemistä varten.
CSS	Cascading Style Sheets. Ohjelmointikieli, jolla kuvaillaan, miltä verkkosivujen eri komponenttien tulisi näyttää.
NoSql	Not only SQL. Ei-relaatiomainen tietokanta.
TDD	Test-driven development. Testivetoinen kehitysmenetelmä jossa luodaan testitapauksia ennen kehitettävää toiminnallisuutta.
BDD	Behavior-driven development. Käyttäytymislähtöinen kehitysmenetelmä jossa luodaan ominaisuudet ennen kehitettävää toiminnallisuutta.
CI	Continuous integration. Jatkuva integraatio, eli kehitysmenetelmä, jossa usean kehittäjän täytyy integroida koodia jaetussa versionhallinnassa useita kertoja päivässä.
IDE	Integrated development environment. Ohjelmointiympäristö on työkalu, jolla voi suunnittella ja ohjelmoida.
YAML	YAML Ain't Markup Language. Ihmisluettava merkintäkieli, jota käytetään usein konfiguraatio-tiedostoissa.
JVM	Java virtual machine. Virtuaalikone Javan ajoympäristöksi.
REPL	Read–eval–print loop. Interaktiivinen kehitysympäristö, jossa kehittäjä antaa komentoja tai komentosarjoja ja saa heti palautteen.
SSH	Secure Shell. Salattu verkkoprotokolla.
CQRS	Command Query Responsibility Segregation. CQRS muodostuu sanoista Command Query Responsibility Segregation ja sillä tarkoitetaan sovellusarkkitehtuurin rakennetta, jossa komento- ja kyselymallit ovat eroteltuna toisistaan.
IaaS	Infrastructure as a Service. IaaS-pilvipalvelumallissa palveluntarjoaja tarjoaa asiakkaalle käyttöön palvelimen pilvessä.

PaaS	Platform as a Service. PaaS-malli tarkoittaa, että palveluntarjoaja tarjoaa alustan, jolla asiakkaat voivat kehittää, suorittaa ja hallita sovelluksia ilman monimutkaisia infrastruktuurisäädöksiä.
SaaS	Software as a Service. SaaS on palvelumalli, jossa valmista sovellusta pääsee käyttämään internetin välityksellä.
FaaS	Function as a Service. Palvelimeton arkkitehtuuri eli Function as a Service(FaaS) tarkoittaa sovellusta, joka on riippuvainen kolmannen osapuolen palveluista tai koodista, joka ajetaan lyhytaikaisissa konteissa.

1 Johdanto

Tämän insinöörityön tarkoituksena oli toteuttaa asiakkaalle uuden yritysmarkkinointi-verkkokaupan pienin mahdollinen toteutus tuotteesta, jolla voidaan testata tuotteen toimivuutta asiakkailla. Tällaista toteutusta kutsutaan Minimum Viable Product -toteutukseksi. Asiakkaan käytössä oleva verkkokauppa tarvitsi uudistusta, sillä se oli jäänyt ajasta jälkeen. Uuden verkkokaupan toteutus moderneilla teknologioilla ja käytännöillä varmistaisi toimivan sekä responsiivisen kokonaisuuden. Projekti oli eräänlainen kokeilu, ja päätimme asiakkaan kanssa, että verkkokaupasta tehdään Minimum Viable Product -toteutus.

Kilpailu yritysmarkkinointialalla on kovaa, ja asiakas halusi hakea uutta markkinaetua uudella verkkokaupalla. Minimum Viable Product -toteutuksella pystyimme nopeasti tuottamaan ja julkaisemaan toimivan tuotteen. Näin asiakas pääsi mahdollisimman nopeasti näkemään uuden tuotteen ja sen vaikutuksen myyntiin. Päätimme myös asiakkaan kanssa, että sivusta tulisi Single-page application, eli verkkosivu toimisi yhdeltä sivulla. Kiireellisen aikataulun takia sivun ominaisuudet rajattiin mahdollisimman tarkasti, niin että pystyimme keskittymään sivun olennaisimpiin toiminnallisuuksiin. Verkkokaupan määritykseen kuului sovelluksen aloitussivu, tuotteiden rajaus kategorioiden mukaan, tuotesivut ja tilausten jättäminen.

Ainoa asiakkaan vaatimus oli, että käyttäisimme Azure-pilvipalvelua kehitys-tuotanto-alustana. Muuten saimme vapaat kädet valita sovelluksen kehittämiseen käytettävät teknologiat.

Front end -teknologioissa päädyimme käyttämään Reagent-kirjastoa, joka on ClojureScriptin ja Reactin välinen rajapinta. Sivuston HTML luotiin käyttäen Clojuren Hiccup-kirjastoa. CSS:n luomiseen taas käytimme Clojurelle tehtyä Garden-kirjastoa.

Sovelluksen back end toteutettiin käyttäen serverless-arkkitehtuurimallia, jossa back end -toiminnallisuus suoritetaan käyttäen Azuren funktioita. Funktioiden ohjelmoimiseen käytimme ClojureScriptiä. Tuotedataa säilytimme NoSql-tietokannassa Azuren DocumentDB:ssä.

2 Sovelluskehitysmenetelmät, Lean Startup ja testaus

Lean Startup on Eric Riesin kehittämä metodologia yritysten ja tuotteiden kehittämiseen. Keskeisin hypoteesi Lean Startup -metodologiassa on, että startup-yritykset sijoittavat aikansa iteratiivisesti rakentaen tuotteita tai palveluita tavoittaakseen asiakkaiden tarpeet. Lean startup on monipuolinen ja skaalautuva, eikä se ole sidottu mihinkään tiettyyn toimialaan, yrityssektoriin tai yrityskokoon. [1, s. 8-9.]

Aikaisessa vaiheessa tulimme siihen johtopäätökseen asiakkaan kanssa, että projekti olisi eräänlainen kokeilu. Päämääränä oli lisätä asiakkaalle myyntiä ja liikevaihtoa. Tapa, millä tähän päästään, ei ollut konkreettisesti selvillä suunnitteluvaiheessa. Tästä johtuen tulimme siihen johtopäätökseen, että Lean Startup olisi täydellinen metodologia tähän projektiin, sillä näin pystyisimme rakentamaan tuotteen, jonka pohjalta voisimme tehdä mittauksia, joiden pohjalta taas oppimisiimme jotain uutta, jota voidaan taas käyttää rakentamiseen ja jatkokehittämiseen.

2.1 Rakenna-Mittaa-Opi

Startup-yrityksen ydintehtävä on muuttaa ideoita tuotteiksi tai palveluiksi. Näiden tuotteiden tai palveluiden rakentamisesta saadaan palautetta, jota voidaan mitata. Kun palautetta mitataan, voidaan oppia jotain uutta, jonka perusteella voidaan taas rakentaa jotain uutta.

Lean Startupissa ensimmäinen kokeilu tarkoittaa ensimmäistä tuotetta. Jos tämä kokeilu onnistuu, voidaan se ottaa käyttöön osalle tuotteen tai palvelun käyttäjistä. Tämä on siis pienin mahdollinen testattava toteutus tuotteelle eli Minimum Viable Product. Ensimmäisen toteutus on siis Lean Startupin ensimmäinen ydinresurssi, eli rakentaminen.

Kun asiakkaat käyttävät yrityksen tuotetta tai palvelua, siitä syntyy kvalitatiivista ja kvantitatiivista palautetta. Lean Startupin toinen ydinresurssi on mittaaminen, eli saatu palaute voidaan mitata.

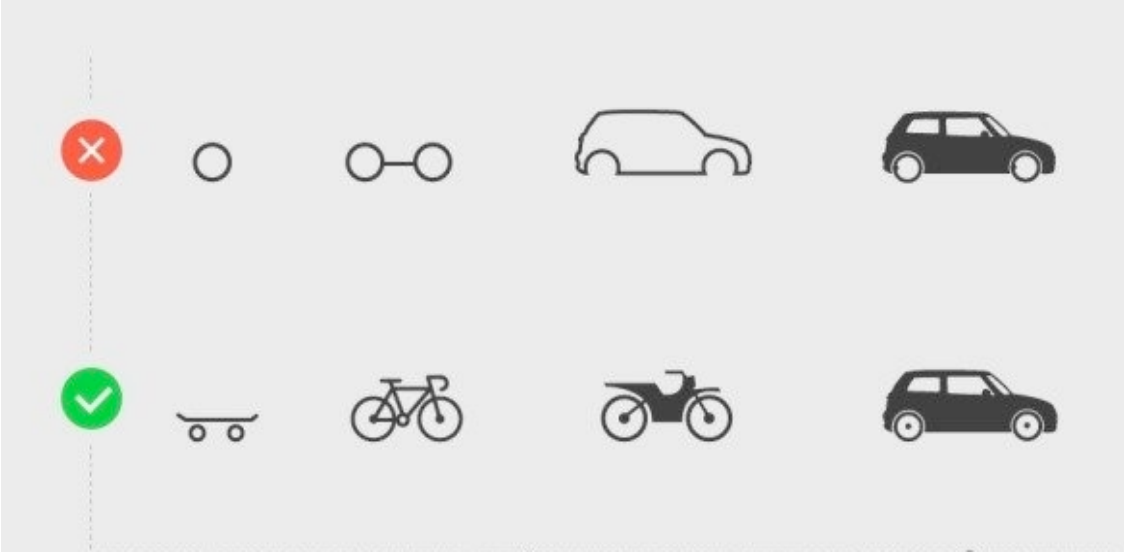
Lean Startupin kolmas ydinresurssi on oppiminen. Mitatusta palautteesta voidaan oppia uutta, jota voidaan taas käyttää uuden rakentamiseen. [1, s. 75-78.]

Koska projekti oli kokeilu ja aikataulu lyhyt, oli luonnollista tehdä sivustosta Minimum Viable Product -toteutus. Näin voisimme keskittyä tuotteen testaamiseen mahdollisimman nopeasti asiakkailla ja näkisimme, onko tuote kannattava ja arvoa tuottava.

2.2 MVP, Minimum Viable Product

Termillä Minimum Viable Product tarkoitetaan siis pienintä mahdollista toteutusta tuotteesta, jolla voidaan testata tuotetta asiakkailla. MVP-tuote on siis mahdollisimman yksinkertainen versio tuotteesta, joka voidaan julkaista. MVP-tuote perustuu kolmeen avainominaisuuteen:

- Tuote on tarpeeksi arvokas, että ihmiset suostuvat ostamaan tai käyttämään tuotetta.
- Tuote osoittaa itsensä tarpeeksi hyödylliseksi niin, että tuotteen aikaisetkin käyttöönottajat säilyvät.
- Tuote tuottaa palautetta tulevaisuuden kehitystä varten. [2]



Kuva 1. Pienin mahdollinen toteutus tuotteesta, jolla voidaan testata tuotetta https://www.linkedin.com/pulse/your-ultimate-guide-minimum-viable-product-kunal-naik

Kuvassa 1. voidaan nähdä MVP-tuotteen kehityskaari. Ensimmäistä tuotetta voidaan vasta testata ja käyttää, kun tuote on valmistunut, eli se tuottaa arvoa hyvin myöhäisessä

vaiheessa kehityskaartaan. Toista tuotetta voidaan taas testata tuotteen jokaisessa kehitysvaiheessa, ja se tuottaa alusta lähtien arvoa.

2.2.1 MVP:n muodostaminen käyttäjätarinoita käyttäen

Käyttäjätarina on määrittelytekniikka. Nimensä mukaistesti sillä voidaan kertoa tarina käyttäjästä, joka käyttää tuotetta tai palvelua ja mikä merkitys tuotteella tai palvelulla on hänelle. Verrattuna perinteiseen ominaisuuksien kehittämiseen on käyttäjätarinoilla nopeampi sykli tarjota arvoa ja palautetta, sillä ominaisuudet eivät yksin tarjoa arvoa eikä palautetta. Käyttäjätarinoita käyttäen on kehittäjillä myös selkeämpi kokonaisuuskuva projektista.

Käyttäjätarinat perustuvat siihen, että *KUKA* pystyy tekemään *MITÄ* ja *MIKSI*. Käyttäjätarinat kirjoitetaan myös selkeästi ei-teknisellä kielellä. On siis erittäin tärkeää, että käyttäjätarinat kirjoitetaan hyvin. Bill Wake kehitti tähän tarkoitukseen ”INVEST”-muistitekniikan, jossa jokaisella kirjaimella on merkitys:

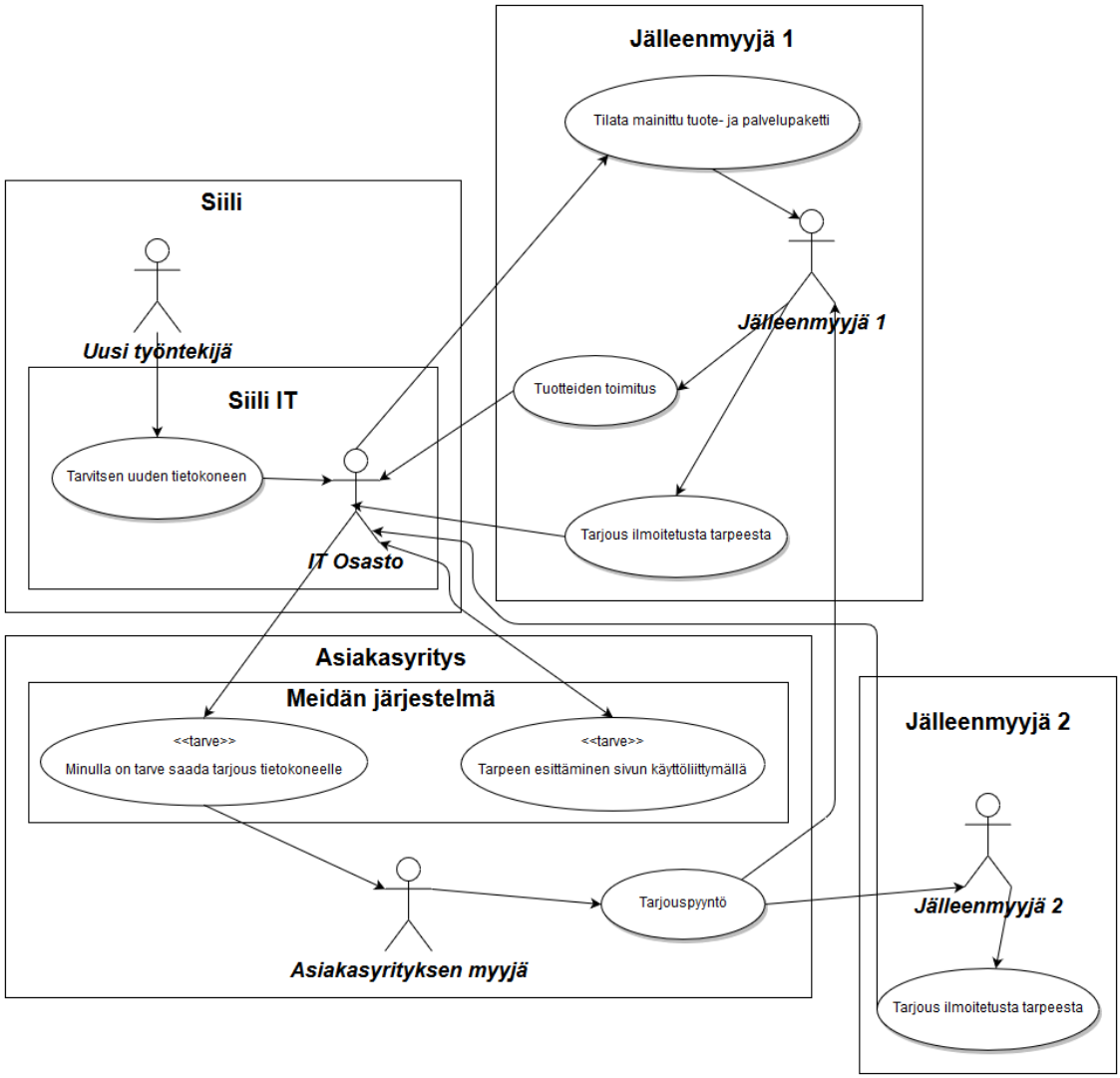
- I, itsenäinen. Käyttäjätarina ei saa olla riippuvainen muista käyttäjätarinoista.
- N, neuvoteltavissa. Käyttäjätarinoiden on aina jätettävä tilaa keskustelulle, jotta niistä voidaan neuvotella.
- V, valuoitavissa. Käyttäjätarinan on tuotettava arvoa käyttäjälle.
- E, estimoitavissa. Käyttäjätarinat on aina oltava helposti arvioitavissa.
- S, small eli pieni. Käyttäjätarinoiden on oltava tarpeeksi pieniä, että ne ovat helppo suunnitella ja priorisoida.
- T, testattavissa. Käyttäjätarinoiden on tarjottava tarpeeksi tietoa, että niiden perusteella voidaan aloittaa testaaminen. [3]

Projektin ensimmäisellä viikolla pohdimme asiakkaan kanssa heidän tarpeita ja määrittelimme mahdollisimman kattavan käyttäjätarinakokoelman, jotta voisimme löytää koh-

dat heidän prosessista, jossa voisimme auttaa. Asiakasyritys toimi siis eräänlaisena tukkurina, jonka toimintamalli perustui osittain yritysten, eli ostajien ja jälleenmyyjien yhdistämiseen. Esimerkkitapauksena käytimme tapausta, jossa meidän yritykseen on tullut uusi työntekijä, joka tarvitsee uuden tietokoneen itselleen. Uusi työntekijä ottaa yhteyttä IT-osastoon ja hän pyytää, saisiko hän uuden tietokoneen itselleen. Yritysemme IT-osasto haluaisi pystyä esittämään tarpeensa verkkosivulle, jotta tarjouspyynnön saaminen olisi mahdollista. Verkkosivun kautta lähtisi asiakasyritykselle viesti, että sivun käyttäjällä on tarve saada tarjouspyyntö tietystä tuotteesta. Näin asiakasyrityksen myyjä voisi ottaa yhteyttä tuotteen jälleenmyyjiin, jotka voisivat lähettää halutusta tuotteesta tarjouksen yritykselle.

Edellisen esimerkkitapauksesta päädyimme seuraaviin käyttäjätarinoihin:

- Uutena työntekijänä minulla on tarve saada uusi tietokone.
- IT-Osastona minä haluan esittää tarpeeni käyttöliittymällä.
- IT-Osastona minulla on tarve saada tarjous tietokoneelle.
- Asiakasyrityksen myyjänä minulla on tarve saada tarjouspyyntö tietokoneelle.
- Jälleenmyyjänä haluan antaa tarjouksen ilmoitetuista tarpeista.
- IT-Osastona haluan tilata tarjouksessa mainitut tuotteet ja palvelut.
- Jälleenmyyjänä haluan toimittaa tuotteet.



Kuva 2. Käyttäjätarinat kuvitettuna

Kuvassa 1 on käyty läpi käyttäjätarinat, joihin päädyimme. Näistä tarinoista oli kuitenkin valittava tietyt tarinat, joita lähtisimme toteuttamaan. Kuvassa 1 on merkattu (”<<tarve>>”) kaksi käyttäjätarinaa, jotka päätimme toteuttaa, eli ”Minulla on tarve saada tarjous” ja ”Tarpeen esittäminen sivun käyttöliittymällä”. Nämä käyttäjätarinat muodostivat meidän Minimum Viable Product -tuotteen.

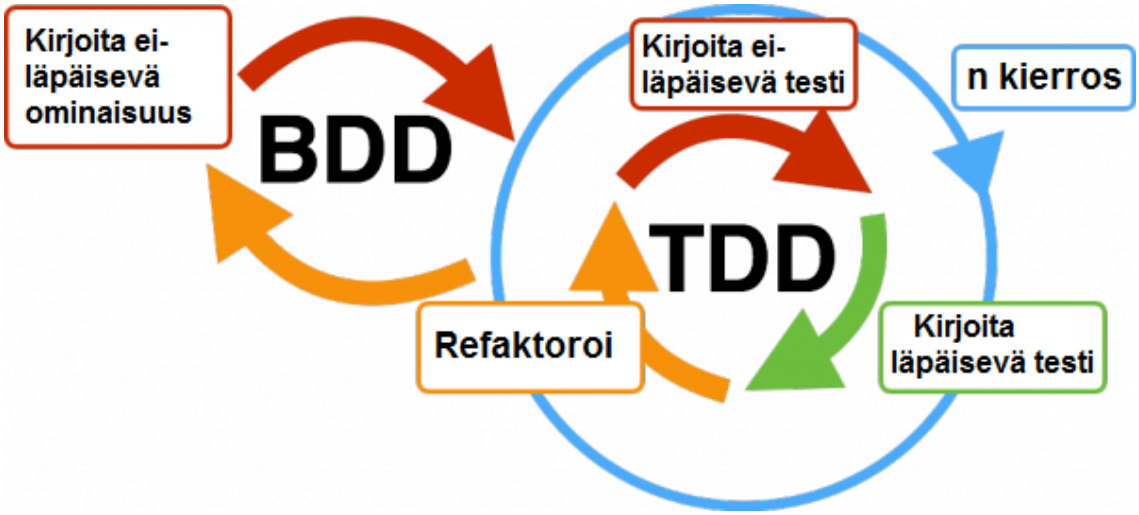
2.3 Testaus- ja käyttäytymislähtöinen ohjelmistokehitys

Ohjelmistojen testaus on ohjelmistokehityksessä toimintatapa, jossa on tarkoitus seurata ja tutkia ohjelmistojen toimivuutta ja laatuominaisuuksia. Ohjelmistotestaus tarjoaa myös

objektiivisen sekä puolueettoman näkemyksen jokaiselle asiaankuuluvalle sidosryhmälle, niin ohjelmoijille, suunnittelijoille, käyttäjille kuin myös liiketoiminnasta vastaaville. Testaaminen nähdään myös iteratiivisena prosessina, sillä yhden vian korjaaminen voi paljastaa useamman muun vian ohjelmassa.

Testivetoiseksi ohjelmistokehitykseksi kutsutaan pragmaattista menetelmää, jossa luodaan testitapauksia ennen haluttua toiminnallisuutta. Tarkoituksena on, että alussa kirjoitetut testit eivät mene hyväksytysti läpi. Kun testitapaukset on tehty, voidaan aloittaa itse ohjelmointi. Kun uusi toiminnallisuus läpäisee aikaisemmin kirjoitetut testit, voidaan taas aloittaa seuraavien testitapauksien kirjoittaminen. Näin saadaan lopuksi muodostettua selkeä ja hyvin testattu ohjelmisto, jota voidaan helposti ylläpitää ja seurata.

Testivetoisesta ohjelmointikehyksestä kehitettiin 2000-luvun vaihteessa uusi selkeämpi kehitystekniikka, jonka tarkoitus oli keskittyä enemmän luotavan toiminnallisuuden käyttäytymiseen kuin pelkkään testaamiseen. Dan North nimesi vuonna 2003 tämän kehitystekniikan käyttäytymislähtöiseksi ohjelmistokehitykseksi.



Kuva 3. Testivetoinen ja käyttäytymislähtöinen testaus https://az184419.vo.msecnd.net/sauce-labs/blog-images/bdd-workflow-600x268.png

Kuvassa 2 nähdään, kuinka TDD, eli testivetoinen kehitys toimii osana BDD:tä eli käyttäytymislähtöistä testausta. Ensiksi kirjoitetaan siis ominaisuus, joka ei toimi. Tämän jälkeen kirjoitetaan testejä, jotka eivät mene läpi. Kun saadaan testit menemään läpi, saadaan ominaisuuskin menemään läpi, jonka jälkeen voidaan aloittaa seuraava ominaisuus.

Käyttäytymislähtöisessä ohjelmistokehityksessä voidaan käyttää Gherkin-ohjelmointikieltä testitapausten ja käyttötapausten kirjoittamiseen. Käyttötapausten määrittäminen Gherkin-ohjelmointikielellä on tehty ihmiselle mahdollisimman helposti luettavaksi ja ymmärrettäväksi niin, että ei-teknisetkin ihmiset saadaan osallistumaan prosessiin. Käyttötapauksia kutsutaan ominaisuuksiksi ja näillä voidaan korkealla tasolla selittää, mitä kyseisessä tapauksessa halutaan toteuttaa. Ominaisuudet voidaan purkaa pienempiin osiin ja näitä pienempiä osia kutsutaan skenaarioiksi. Skenaariot kuvaavat tarkemmin yhtä tiettyä toimintoa, joka halutaan saavuttaa. Skenaariot perustuvat kolmeen osaan Given, When ja Then. Given-osassa määritetään skenaarion edellytykset. When-osassa taas määritetään toiminto, mikä skenaariossa tehdään. Then-osassa eritellään haluttu lopputulos. [4]

Seuraavana esitellään esimerkki ominaisuudesta, jossa tehdään tuotetilaus verkkokaupasta.

> Käyttäjä haluaa tehdä tilauksen verkkokaupasta. Käyttäjä etsii haluamansa tuotteet, jotka hän haluaa tilata. Käyttäjä lisää haluamansa tuotteet ostoskoriin. Käyttäjä vahvistaa ostoskorin sisällön ja maksaa tuotteet. Käyttäjä näkee tilausvahvistuksen.

Skenaario 1: Käyttäjä etsii haluamansa tuotteet.

Given: Käyttäjä on tuotehakusivulla.

When: Käyttäjä suorittaa tuotehaun.

Then: Käyttäjä näkee haluamansa tuotteen.

Skenaario 2: Käyttäjä lisää tuotteen ostoskoriin.

Given: Käyttäjä on tuotesivulla.

When: Käyttäjä painaa ’lisää ostoskoriin’-nappia.

Then: Tuote on lisätty ostoskoriin.

Skenaario 3: Käyttäjä vahvistaa ostoskorin sisällön ja maksaa tuotteet.

Given: Käyttäjä näkee ostoskorin sisällön ja maksu-napin.

When: Käyttäjä painaa ‘mene maksamaan’-nappia.

Then: Käyttäjä siirretään verkkopankkiin maksamaan.

Skenaario 4: Käyttäjä näkee tilausvahvistuksen.

Given: Käyttäjä on suorittanut verkkomaksun.

When: Käyttäjä siirretään verkkopankista, kaupan verkkokauppaan.

Then: Käyttäjälle näytetään tilausvahvistus.

Esimerkkitestitapaukset eivät käytännössä vielä testaa mitään, mutta näiden perusteella on huomattavasti helpompi kirjoittaa itse testit. Projektissa käytimme siis käyttäytymislähtöistä ohjelmistokehitystä ja käymme työssä myöhemmin läpi, miten Gherkin-testitapauksiin yhdistetään Clojuren testaustoiminnallisuus.

3 Kehitystyökalut

Tässä luvussa käsitellään projektissa käytettyjä työkaluja sekä teknologioita ja näytetään esimerkkejä vastaavasta toteutuksesta.

3.1 Versionhallintajärjestelmä ja Git

Versionhallintajärjestelmä on ohjelmistotyökalu, joka on tarkoitettu ohjelmistokehittäjille. Versionhallintajärjestelmillä pystytään hallitsemaan lähdekoodin muutoksia ajan mittaan. Meidän projektissa päätimme käyttää versionhallintajärjestelmää nimeltä Git.

Git on hajautettu versionhallintaohjelmisto, joka pitää kirjaa tiedostojen muutoksista. Se on Linus Torvaldsin luoma, alun perin Linuxin ytimen kehittämistä varten, ja se julkaistiin huhtikuussa 2005. Git on julkaistu GNU General Public -lisenssin alla ja sitä voi käyttää graafisella käyttöliittymällä, komentotulkista tai IDE:stä.

Git perustuu kolmeen tilaan, jotka ovat työhakemisto, pysyvä alue ja .git-hakemisto eli arkisto. Työhakemistolla viitataan käyttäjän paikalliseen työhakemistoon. Kun käyttäjä on tehnyt muutoksia työhakemistoon, hän voi lisätä muutokset pysyvälle alueelle. Kun pysyvän alueen muutoksiin ollaan tyytyväisiä, voidaan muutokset sitouttaa arkistoon. Aina kun arkistoon sitoutetaan muutos, otetaan tilasta tilannekuva. Tilannekuvilla voidaan hallita koodimuutoksia ja kokonaisuutta. [5]

3.2 Docker ja konttiteknologia

Docker on avoimen lähdekoodin projekti, jolla voidaan helposti automatisoida sovellusten käyttöönotto, niin sanottujen konttien sisällä. Docker-kontit paketoivat osan sovellusta omaan tiedostojärjestelmään, joka myös sisältää kaiken, mitä sovelluksen suorittamiseen tarvitaan. Näin voidaan varmistua siitä, että sovellus voidaan aina suorittaa eikä versiokonflikteja tule. Docker-kontit vaativat isäntäkoneen, missä toimia. Dockerin konttijärjestelmä ovat myös kevyempi kuin virtuaalikoneet, sillä jokainen kontti voidaan suorittaa isäntäkoneella.

Docker-kontin sisälltö ja suoritustapa voidaan määrittää Dockerfile-tiedostossa. Näitä kutsutaan kuviksi. Kuvia voidaan myöhemmin käyttää uudestaan kontin pystyttämiseen ja niiden jakamista varten on perustettu dockerhub.com, josta mekin saimme käyttöömme tarvittavat Dockerfile-tiedoston. [6]

Seuraavassa esimerkkikoodissa on meidän jatkuvassa integraatiossa käyttetty Dockerfile-tiedosto.

```
FROM ubuntu:14.04
ADD https://github.com/Yelp/dumb-init/releases/down-
load/v1.0.2/dumb-init_1.0.2_amd64 /usr/bin/dumb-init
RUN chmod +x /usr/bin/dumb-init

RUN apt-get update -y && \
    apt-get upgrade -y && \
    apt-get install -y ca-certificates wget apt-
transport-https vim nano && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

RUN echo "deb https://packages.gitlab.com/run-
ner/gitlab-ci-multi-runner/ubuntu/ `lsb_release -cs`
```

```
main" > /etc/apt/sources.list.d/runner_gitlab-ci-
multi-runner.list && \
    wget -q -O - https://packages.gitlab.com/gpg.key |
apt-key add - && \
    apt-get update -y && \
    apt-get install -y gitlab-ci-multi-runner && \
    wget -q https://github.com/docker/machine/re-
leases/download/v0.7.0/docker-machine-Linux-x86_64 -O
/usr/bin/docker-machine && \
    chmod +x /usr/bin/docker-machine && \
    apt-get clean && \
    mkdir -p /etc/gitlab-runner/certs && \
    chmod -R 700 /etc/gitlab-runner && \
    rm -rf /var/lib/apt/lists/*

ADD entrypoint /
RUN chmod +x /entrypoint

VOLUME ["/etc/gitlab-runner", "/home/gitlab-runner"]
ENTRYPOINT ["/usr/bin/dumb-init", "/entrypoint"]
CMD ["run", "--user=gitlab-runner", "--working-direc-
tory=/home/gitlab-runner"]
```

Esimerkkikoodi 1. Meidän Gitlab runnerin käyttämä Dockerfile-tiedosto

Esimerkki Dockerfile-tiedosto koostuu seuraavista komennoista:

- FROM, Pakollinen kommento, jossa määritetään, mitä käyttöliittymää ja versiota halutaan käyttää
- ADD, Lisäys-komennolla, voidaan kopioida tiedosto paikallisesti tai ladata se verkosta
- RUN, Suorita-komento, jolla suoritetaan eri komennot kontin shellissä
- VOLUME, Hakemisto-komennolla, voidaan määrittää käytettävä hakemisto, joka on kontin ulkopuolella isäntäkoneella
- ENTRYPOINT, Aloitus-komento, joka suoritetaan aina kun kontti käynnistyy
- CMD, Oletus-suoritus komento, jossa määritetään mitä suoritetaan aloitus-komennon yhteydessä

Esimerkkikoodi 1 Dockerfile-tiedosto alkaa sillä, että määritetään FROM-komennolla käyttöjärjestelmäksi Ubuntu ja versioksi 14.04. ADD-komennolla kopioidaan verkosta ohjelma nimeltä Dumb-init, jolla hallitaan prosessien valvontaa. RUN-komennolla määritetään ensin Dumb-init-tiedoston oikeudet, jonka jälkeen seuraavilla RUN-komennoilla ladataan ja asennetaan muut tarvittavat ohjelmat ja asennetaan ne ja mahdolliset päivitykset. ADD-komennolla lisätään kotihakemisto ”entrypoint”:ksi ja annetaan RUN-komennolla sille tarvittavat oikeudet. VOLUME-komennolla lisätään isäntäkoneen "/etc/gitlab-runner"- ja "/home/gitlab-runner" -hakemistot kontille. Viimeisella komennolla, eli CMD-komennolla määritettiin, että kun kontti käynnistyy, käynnistetään itse Gitlab-runner, jota käytimme jatkuvassa intergraatiossa.

3.3 Jatkuva integraatio ja käyttöönotto

Jatkuva integraatio on kehitysmenetelmä, jossa usean kehittäjän täytyy integroida koodia jaetussa versionhallinnassa useita kertoja päivässä. Jokainen koodimuutos sen jälkeen tarkastetaan automaattisella sovelluksen kääntämisellä, jolloin pystytään huomaamaan ongelmat mahdollisimman aikaisin.

Koska integraatio on niin tiheää, on paljon helpompi korjata virheet, kun ne tulevat heti esille, eikä siten, että ensin täytyy etsiä, mikä hajosi ja milloin. Jatkuva integraatio on myös paljon halvempaa kuin olla ilman. Mitä harvemmin integraatio ja tarkastus tehdään, ongelmien ratkomisen korjaamisen haastavuus kasvaa eksponentiaalisesti. Vaikeiden ongelmien korjaus helposti saa projektin aikataulun horjumaan tai jopa pahimmassa tapauksessa epäonnistumaan kokonaan. [7]

Jatkuva integraatio on siis pakollinen osa mitä tahansa projektia, jossa on useampi henkilö kirjoittamassa koodia. Sama tilanne oli myös meidän projektissa, ja sen vuoksi lisäsimme siihen jatkuvan integraation käyttämällä Gitlab Runner -työkalua. Gitlab runner on avoimen lähdekoodin projekti, jolla ajetaan määritellyt itsemääritellyt tehtävät, kun versionhallintaan tulee muutoksia. [8]

Projektissa loimme docker-kontin, jonne Gitlab runner asennettiin. Docker-kontti puolestaan asennettiin Azureen virtuaalikoneelle, josta kerromme lisää Azure-luvun alla. Tä-

män jälkeen versionhallintaan lisättiin gitlab runner -komentotiedosto, johon määrittelimme tehtävät, joita sen tuli ajaa joka kerta, kun versionhallintaan tuli uusi versio. Gitlab runnerin komentotiedostot kirjoitetaan YAML-merkintäkielellä.

Gitlab runnerille on määritetty tietyt valinnaiset avainsanat, joita voidaan käyttää kirjoittaessa komentotiedostoa.

- image, image-kohdassa määritetään Docker-kuva, mitä halutaan runnerin kontissa käyttää.
- services, services-kohdassa määritetään käytettävät Docker-palvelut.
- stages, stages-kohdassa esitetellään omat kometosarjat, jotka tullaan esittelemään tiedostossa myöhemmin.
- types, vanhentunut vaihtoehto stages-avainsanalle.
- before_script, before_script-kohdassa voidaan määrittää ajettava komentosarja, joka ajetaan ennen muita itse-määrittämiä komentosarjoja.
- after_script, after_scipt-kohdassa määritetään omien komentosarjojen jälkeen ajettava komentosarja.
- variables, variables kohdassa esitellään muuttujat, joita kääntämisessä halutaan käyttää.
- cache, cache-kohdassa määritetään lista tiedostoita, jotka halutaan asettaa välimuistiin jokaisen suorituksen välissä.

```
image: adzerk/boot-clj
services:
  - postgres

before_script:
  - bundle install

after_script:
  - rm secrets

stages:
  - build
```

```
  - test
  - deploy

job1:
  stage: test
  script:
    - execute-script-for-test-job1
  only:
    - master
  tags:
    - docker
```

Esimerkkikoodi 2. YAML-esimerkkitiedosto Gitlab-runnerille https://docs.gitlab.com/ce/ci/yaml/

Ensimmäinen vaihe oli ajaa koodin testit läpi. Läpäistessään ne se lähettää koodit Azuren blob storageen, josta selitetään enemmän Azure-luvun alla.

3.4 Clojure-ohjelmointikieli

Clojure on Rich Hickeyn suunnittelema ohjelmointikieli, joka on eräänlainen murre Lisp-ohjelmointikielestä. Lisp on toiseksi vanhin korkean tason ohjelmointikielistä, joka on laajassa käytössä vielä nykypäivänä. Lisp-kielen ensimmäinen versio kehitettiin vuonna 1958 John McCarthyn toimesta. Lispistä on ajan saatossa tullut muitakin ohjelmointimurteita, mutta Clojure on yksi nykyaikaisimmista ja suosituimmista. [9]

Clojuren päätarkoituksena on ratkaista nykyisten ohjelmointikielien ongelmia. Tällaisia ongelmia on esimerkiksi turha monimutkaisuus. Monissa kielissä on erittäin helppoa tehdä turhaa kompleksisuutta yhtäaikaisten säikeiden, eri datavaatimusten ja toisistaan riippumattomien kirjastojen vuoksi. Esimerkiksi olio-ohjelmoinnissa kaikki ajettavat koodit pakataan eri kerroksiin luokkien alle, joita peritään ja määritellään myöhemmin. Clojuressa tämä estetään tekemällä puhtaita funktioita, jossa funktio ottaa argumentteja ja palauttaa tuloksen pelkästään argumenttien perusteella.

Clojure mahdollistaa monimutkaisten tehtävien tekemisen lyhyesti ja ytimekkäästi. Clojuressa on lähdetty myös tavoittelemaan käytännöllisyyttä. Se on rakennettu JVM:n (Java Virtual Machine) päälle ja asetettu tukemaan kaikkea Java-koodia, sillä se on kehittynyt, nopea ja laajalti käytetty. Tämä sama ajatusmalli on otettu ClojureScriptin, sillä

ClojureScript-koodi kääntyy JavaScript-koodiksi ja siten sitä voi käyttää lähes missä tahansa front end -koodina.

Monissa muissa kielissä on ongelmia myös selkeyden kanssa. Seuraavassa Python-koodiesimerkissä kuvataan tätä ongelmaa Python-ohjelmointikielessä.

```
# Tämä on Python koodia
x = [5]
process(x)
x[0] = x[0] + 1
```

Esimerkkikoodi 3. Python-koodiesimerkki

Esimerkiksi yllä olevasta koodista ei voi olla varma, onko lopputulos 6 vai muuttaako process-metodi x:n arvoa jollain tapaa. Lisäksi on mahdollista, että metodi, jota kutsutaan process-metodista, muuttaa sivuvaikutuksena x:n arvoa. Tähän soppaan voi myös lisätä useamman säikeen, mikä lisää entisestään kompleksisuutta. Tätä varten Clojuressa käytetään muuttumattomia arvoja ja tauluja, sillä se poistaa heti suurimman osan monisäikeisistä ongelmista.

Clojuressa myös syntaksit on muodostettu mahdollisimman yhteneviksi, jotta koodin luettavuus olisi helpompaa ja nopeampaa. Clojuren syntaksi muodostuu siten, että ensimmäisenä asetetaan sulut ja sulkujen sisällä oleva ensimmäinen teksti on funktio, joka suoritetaan ja loput sulkujen sisälle tulevat arvot ovat parametreja, joita lähetetään kutsutulle funktiolle.

```
(defn hello-world [name]
  (let [sentence (str "Hello " name)]
   (println sentence)))

(hello-world "World!")
```

Esimerkkikoodi 4. Clojure Hello world

Esimerkkikoodi 4 on muodostettu yksinkertainen ”hello world” -esimerkki. Ensimmäisellä rivillä on muodostettu uusi funktio komennolla defn. Kentän toisena parametrina on uuden funktion nimi ja kolmanneksi asetetaan hakasulut, jossa ilmoitetaan, miten monta argumenttia funktio ottaa vastaan. Tässä esimerkissä otetaan yksi argumentti ja sidotaan se ”name”-nimiseen muuttujaan. Seuraavalla rivillä alkaa funktion toiminnallisuuden

määrittäminen. Siinä luodaan uusi paikallinen muuttuja let-komennolla, jossa toisena argumenttina tulee vektori. Vektorissa ilmoitetaan muuttujat avain-arvopareilla, ja arvo voi myös muodostua uudesta funktiosta niin kuin tässä tapauksessa. Arvo on luotu tekemällä uusi lukujono, jossa yhdistetään tekstit ”hello ” ja saapunut argumentti. Seuraavalla rivillä kutsutaan println-komentoa, joka tulostaa sitten paikallisen muuttujan. Funktion ulkopuolella on sitten esimerkki, kuinka tätä funktiota kutsuttaisiin. Clojuressa on myös tärkeää, että ne funktiot, joita kutsutaan, täytyy olla määritelty ennen kutsua. [10, s.3-24.]

Projektin ympäristöjen asettamisen jälkeen meidän täytyi valita ohjelmointikieli, jolla tekisimme markkinointiverkkokaupan Azure-pilveen. Halusimme myös palveluiden olevan nykyaikaisia ja mahdollisimman tehokkaita. Koodin täytyi myös olla mahdollisimman toimivaa ja nopeasti kirjoitettavaa tiukan aikataulun vuoksi. Clojure oli näiden syiden vuoksi selkeä vaihtoehto, kuinka päästä eroon olio-ohjelmoinnin aiheuttamista ongelmista helposti, mutta saada silti laaja skaala eri alustoja käyttöön ilman rajoitteita.

3.4.1 REPL-koodinsuoritustyökalu

REPL eli read-eval-print loop on interaktiivinen ohjelmointikehitysympäristö, joka ottaa yksittäisiä käyttäjäsyötteitä sisään, ajaa ne, ja lopuksi tulostaa tuloksen käyttäjälle. [11 s.76.]

REPL on tehokas työkalu Clojuren kehityksessä. Sen avulla kehittäjät voivat nopeasti nähdä, toimiiko heidän kirjoittamansa koodi oikein. Suurin osa Lisp- ja Clojure-ohjelmoijista käyttää jonkinlaista REPL:iä heidän koodinsa testaamiseksi ja kehittämiseksi.

Konseptissa REPL on Secure Shellin (SSH) kaltainen. SSH pystyy yhdistämään etäpalvelimelle, kun taas Clojure REPL voi samaan tapaan keskustella juoksevan Clojure prosessin kanssa. Tästä tehokkaan tekee se, että REPL voidaan yhdistää suoraan pystyssä olevaan tuotantokoodiin ja muokata sitä samalla, kun se suoritetaan. [12]

Tässä projektissa REPL oli käytössä lähinnä kehityksen aikana testaamaan, että luodut funktiot palauttivat oikeaa dataa esimerkiksi, kun tuotettiin uusi funktio, jonka piti palauttaa yhdistetty lista arvoista, joita sinne lähetettiin. Kun funktio oli valmis, niin se syötettiin REPL:iin ja kutsuttiin sitä tekaistulla datalla, joka oli oikeanmuotoista. Tämän jälkeen näki heti, palautuiko sieltä lista oikeassa muodossa.

3.4.2 Clojuren historia

Rich Hickey aloitti Clojuren kehittämisen vuonna 2005 ja käytti siihen kaksi ja puoli vuotta, ennen kuin julkisti sen ensimmäisen version. Tämän jälkeen kehitys muuttui yhteisövetoiseksi ja sitä hallinnoidaan Clojure-yhteisön nettisivulta.

Versio	Julkaisupäivä	Oleellisimmat lisätyt muutokset
	16.10.2007	Alustava julkinen julkaisu
1.0	04.05.2009	Ensimmäinen vakaa julkaisu
1.1	31.12.2009	Futures-komentotyyppi
1.2	19.08.2010	Protokollat
1.3	23.09.2011	Paranneltu primitiivi tuki
1.4	15.04.2012	Lukijan literaalit
1.5	01.03.2013	Supistajat
1.6	25.03.2014	Java-rajapinnan parannellut algoritmit
1.7	30.06.2015	Koottavat algoritmimuunnokset
1.8	19.01.2016	lisä merkkijonofunktiot, suora yhdistäminen, palvelimelle yhdistäminen
1.9	*Tulevaisuudessa*	

Taulukko 1. Clojuren versiohistoria [13]

Clojure oli kielenä jo monta vuotta ollut valmiina ja sitä oli käytetty monissa isoissa projekteissa. Totesimme Clojuren olevan tarpeeksi varttunut kieli käyttöönottoa varten, eikä meidän tarvitse pelätä, ettei se kykene kaikkeen tarvittavaan.

3.5 ClojureScript-ohjelmointikieli

ClojureScript on ohjelmointikieli verkkosovelluksien tekoon. ClojureScriptin on tehnyt Cognitect, joka on Rich Hickeyn perustama yritys. ClojureScriptin ensimmäinen versio julkaistiin vuonna 2012. ClojureScript on rakennettu niin, että siinä on käytetty pohjana Clojurea, mutta se kääntyy JavaScriptille ja sen kääntäjä käyttää Google Closurea.

Google Closure on Googlen tekemä JavaScript-optimoija, jonka avulla voi saada JavaScript-koodin lataamaan ja suoriutumaan nopeammin. Closure jäsentää JavaScript-

koodin, analysoi sen, poistaa käyttämättömän koodin ja lopulta uudelleenkirjoittaa ja minimoi koodin. [14]

ClojureScript-kehittäjät näkivät, että JavaScript-kielessä on paljon ongelmia, mutta se on kuitenkin todella laajalle levinnyt kieli. Sitä ei ole suunniteltu suuria sovelluksia varten, mutta silti se on todella laajasti levinnyt kieli monessa järjestelemässä. Tämän vuoksi toteutettiin ClojureScript, joka on moderni, vankka ja tehokas kieli.

ClojureScript eroaa myös Clojuresta, sillä ClojureScriptistä puuttuu ajonaikainen koodin kääntäminen. ClojureScript ei myöskään ole JavaScript Clojure syntaksilla. Sillä on sama semantiikka kuin Clojurella. [15]

Clojurea on päädytty käyttämään pohjana, sillä se tuo myös JavaScript-maailmaan samat hyödyt kuin Java-maailmaan. JavaScriptin laajuus on vailla vertaa verkkosovelluksissa. Tämän vuoksi ClojureScript-ohjelma käännetään JavaScript-koodiksi, jotta sitä voidaan käyttää lähes missä vain. ClojureScript-sovelluksien kääntämisessä käytetään Google Closurea, joka on Googlen lanseeraama kirjasto ja kääntäjä, jolla tehdään ohjelman optimoinnista mahdollista. Closuren avulla latausajat saadaan todella pieniksi, jopa niissä tilanteissa, kun on käytössä useita kirjastoja. Nämä kaikki yhdistämällä saadaan yksi tehokkaimmista ohjelmointikielistä verkkoa varten. [16]

3.6 Ohjelmointikielet meidän projektissamme

Koska projektissa oli valittu kieleksi Clojure, niin projektin front end -ohjelmointikieleksi valittiin ClojureScript. Projektin back end toimii Azure-funktioissa, joita kutsuttiin front end -palvelusta REST-rajapintakutsuilla. Azure-funktiot tukevat JavaScriptia, joten näitä pystyimme myös kirjoittamaan käyttäen ClojureScriptia.

Projektissa päädyimme käyttämään Clojurea ja ClojureScript-ohjelmointikieliä näiden kaikkien hyötyjen vuoksi, mitä puhtaalla funktionaalisella ohjelmointikielellä voi saavuttaa. Halusimme myös haastaa itseämme ja kasvattaa osaamistamme useammasta ohjelmointikielestä. Clojuren selkeys oli myös lisäetu, koska kun järjestelmää alettaisiin jatkokehittää, mikä oli suunnitelmissa, olisi vanhan koodin lukeminen helppoa ja nopeaa varsinkin siinä tapauksessa, jos kehittäjät vaihtuisivat.

Sen vuoksi, että järjestelmää alkaisivat jatkokehittää eri henkilöt, ajettiin koodille eri staattisia analyysityökaluja. Näiden avulla koodista saatiin turhia koodinpätkiä pois, esimerkiksi turhia if-lauseita. Se myös varmisti, että koodi oli ulkoasullisesti yhtenevää, jotta koodin luku olisi mahdollisimman helppoa.

3.7 Clojuren ja ClojureScriptin kääntötyökalut

Clojure-lähdekoodi pitää kirjoittamisen jälkeen kääntää binäärikoodiksi, jotta kone osaa ajaa suoritettavan koodin. Clojure-lähdekoodi käännetään tarkalleen ottaen JVM-tavukoodiksi, joten tämä tarkoittaa sitä, että Clojure-koodia voidaan ajaa, missä Java-koodiakin voisi ajaa. ClojureScript-lähdekoodi sen sijaan käännetään JavaScript-koodiksi ja sitä voidaan ajaa missä JavaScript-koodia voisi ajaa.

Kääntötyökaluilla saadaan myös tehtyä automatisointia. Käännösprosesseihin pystyy lisäämään eri vaiheita ja suoritustapoja, esimerkiksi sen perusteella, mihin ympäristöön käännetty binäärikoodi ollaan laittamassa. Myös ulkoisten kirjastojen lataaminen saadaan integroitua käännösautomaatioon. [17]

Lähdekoodin kääntämistä varten on Clojurelle tehty kaksi eri kääntäjää, joiden välillä on väittelyä puolesta ja vastaan. Näiden nimet ovat Boot ja Leiningen.

Näiden kahden kääntötyökalun välillä kävimme pohdintaa, kumpi oli parempi vaihtoehto meitä varten. Näiden työkalujen ominaisuuksia käsitellään kahdessa seuraavassa luvussa ja kolmannessa käydään läpi, mikä oli projektia varten parempi ratkaisu ja miksi.

3.7.1 Leiningen-kääntötyökalu

Leiningen on toinen Clojuren kääntötyökaluista, jolla voidaan kääntää Clojure-sovelluksia yksinkertaisten konfiguraatioiden avulla. Sen on alun perin tehnyt Phil Hagelberg ja tällä hetkellä sitä ylläpitää Jean Niklas L'orange. Leiningenistä löytyy muun muassa

- yksinkertaisen Clojure-projektin luurangon luonti
- ahead-of-time kääntäminen

- riippuvuuksien hallinta
- REPL
- projektin lähdekoodin ja riippuvuuksien pakkaaminen “Uberjar”-tiedostoon. [18]

3.7.2 Boot-kääntötyökalu

Bootin on tehnyt Alan Dipert ja Mich Niskin ja sen ensimmäinen versio julkaistiin vuonna 2013. Lähes koko sovellus on kirjoitettu Clojurella. Boot-projekteissa konfigurointi tapahtuu ”build.boot”-nimisen tiedoston avulla, joka sijaitsee projektin juuressa.[19] Tässä työssä kääntötyökaluksi valittiin Boot, sillä Boot on helpompikäyttöisempi, jos käännöstyö vaatii useita eri vaiheita. [20]

Seuraavaksi oleva koodiesimerkki on build.boot-tiedostosta, jolla muokataan Bootin toimintoja ja lisätään ulkopuolisia kirjastoja. Rivillä kaksi ilmoitetaan Bootille, mistä lähdekoodit projektiin löytyvät. Tässä tapauksessa html-sivut ovat html-kansion alla ja lähdekoodit src-kansion alla.

```
(set-env!
  :resource-paths #{"src" "html"}
  :dependencies '[[adzerk/boot-cljs "1.7.228-1" :scope "test"]
                  [adzerk/boot-cljs-repl "0.3.0" :scope "test"]
                  [adzerk/boot-reload "0.4.5" :scope "test"]
              [pandeiro/boot-http "0.7.1-SNAPSHOT" :scope "test"]
[crisptrutski/boot-cljs-test "0.2.2-SNAPSHOT" :scope "test"]
                  [org.clojure/clojure         "1.7.0"]
                  [org.clojure/clojurescript   "1.7.228"]
                  [com.cemerick/piggieback "0.2.1" :scope "test"]
                  [weasel "0.7.0"          :scope "test"]
               [org.clojure/tools.nrepl "0.2.12" :scope "test"]
        [reagent "0.6.0"]])
```

Esimerkkikoodi 5. Build.boot-tiedoston ulkoiset kirjastot

Esimerkkikoodi 5 sisältää erinäisiä ulkoisia kirjastoja, joita on lisätty projektin riippuvuuksiin. Nämä tiedostot ladataan automaattisesti kääntämistä varten.

```
(require
'[adzerk.boot-cljs      :refer [cljs]]
'[adzerk.boot-cljs-repl :refer [cljs-repl start-repl]]
```

```
'[adzerk.boot-reload    :refer [reload]]
'[crisptrutski.boot-cljs-test  :refer [exit! test-cljs]]
'[pandeiro.boot-http    :refer [serve]])
```

Esimerkkikoodi 6. Build.boot tiedoston riippuvuudet

Esimerkkikoodi 6 sisältää Require-komennolla ladattavat kirjastot; sama kuin Javan import.

```
(deftask testing []
 (merge-env! :resource-paths #{"test"})
 identity)

(deftask auto-test []
 (comp (testing)
       (watch)
       (speak)
       (test-cljs)))

(deftask dev []
 (comp (serve :dir "target/")
       (watch)
       (speak)
       (reload :on-jsload 'app.core/main)
       (cljs-repl)
       (cljs :source-map true :optimizations :none)))

(deftask test []
 (comp (testing)
       (test-cljs)
       (exit!)))

(deftask build []
 (cljs :optimizations :advanced))
```

Esimerkkikoodi 7. Build.boot-tiedoston ajettavia funktioita

Esimerkkikoodi 7 sisältää useita eri boot-komentoja, joita voidaan sitten kutsua boot:n komentorivityökalun kautta. Esimerkiksi ajamalla komentorivillä “boot dev” -komennon boot käynnistää kehitysversion lokaaliin ympäristöön, ja kaikki muutokset pääsevät näkemään saman tien, kun koodia editoi. Kun yhdistää esimerkkikoodit 4 – 6 saa muodostettua kokonaisen Build.boot-tiedoston.

Projektissa käytimme boot-kääntötyökalua, ja tenzingin luoma automaattinen build.boot-tiedosto oli projektin alkuvaiheissa tarpeeksi kattava, eikä tarvinnut mitään ylimääräistä

konfigurointia. Pystyimme kehittämään sovellusta omalla koneella suoraan dev-komennolla, joka käynnisti projektin ”localhost:8080”-osoitteeseen ja päivittyi automaattisesti aina, kun koodiin teki muutoksia.

Kun projektiin lisättiin automaattinen integraatio, täytyi luoda uusi funktio, joka osasi lähettää koodit haluttuun pilviympäristöön. Toinen muutos, mitä jouduimme projektin aikana tekemään build.boot-tiedostoon, oli eri ulkoisten kirjastojen tuonti projektiin.

Siinä vaiheessa, kun ohjelmallinen toiminnallisuus oli valmistunut, koodiin kääntötyökaluun lisättiin vielä staattinen ohjelma analyysityökalu. Se käy kaiken kirjoitetun koodin läpi ja tarkastaa sen ulkoasun ja kertoo, miten siitä tehdään yhtenevä. Se myös tarkastaa koodista käyttämättömät osat sekä turhat kompleksisuudet. Esimerkiksi se poistaa turhat if-lauseet koodista. Analyysityökalulla saimme siistittyä koodit viimeiseen ohjelmistoversioon.

3.7.3 Kääntötyökalujen yhteenveto

Kääntötyökalujen välillä käytiin vertailua siitä, kumpi on sopivampi meidän tarpeisiin Leiningen vai Boot. Boot osoittautui käytännöllisemmäksi kääntötyökaluksi niissä tilanteissa, joissa oli monta kääntövaihetta, kuten tässä projektissa. Käännösprosessissa täytyi kääntää ClojureScript-koodit JavaScriptiksi, Garden koodit CSS:ksi, lähettää frontend-koodit ja Azure-funktio-koodit eri sijanteihin, sekä luoda uusi funktio, joka osaa lähettää koodit Azureen.

Projektissa lopulta tavallista Clojurea oli pelkästään Boot-automaatio skripteissä ja CSS:n luonnissa. Loput koodit projektiin kirjoitettiin ClojureScriptillä, sillä selaimessa ajettava koodi vaati JavaScriptia ja Azure-funktiot tukivat JavaScriptia.

3.8 HTML ja Hiccup verkkosivuteknologioina

HTML eli HyperText Markup Language on standardoitu kieli verkkosivujen ja verkkoohjelmien tekemistä varten. Selaimet lataavat HTML-sivut verkkopalvelimelta ja piirtävät ne multimediaverkkosivuina näytölle. HTML-sivussa kuvaillaan, mihin kohtiin komponentit tulevat paikalle verkkosivuilla.

HTML-sivuilla elementit laitetaan paikoilleen käyttämällä “tägejä”, esimerkiksi . Huomiona tähän, että elementti päättyvät lähes samanlaiseen merkkiin kuin mistä ne alkavat, erona niissä on lisättynä “/“. Elementtien sisältö tulee alku- ja loppumerkin välille. [21]

Projektissa käytimme Reagenttia, jossa UI-komponentit muodostettiin Clojuren Hiccup-kirjaston mallisella syntaksilla. Reagentissa HTML-elementtejä muodostetaan seuraavalla tavalla:

```
[:div
          [:h1 "Tervetuloa verkkokauppaan"]
          [:p "Löydä nopeimmat ja parhaimmat ratkaisut meidän
nopeasta ja luotettavasta verkkokaupasta helposti ja vaivatto-
masti ilmaisella toimituksella."]]
```

Esimerkkikoodi 8. Reagent/hiccup-koodiesimerkki.

HTML-koodin osaaminen on pakollista lähes kaikissa verkkoprojekteissa. Reagentia käytettäessä on ymmärrettävä, miten HTML-koodi toimii, jotta osaa kirjoittaa myös Hiccup-syntaksilla HTML-koodia.

3.9 Tyylittely CSS:llä ja Gardenilla

CSS Eli Cascading Style Sheets on kieli, jolla kuvaillaan, miltä verkkosivujen eri komponenttien tulisi näyttää. [22]

CSS on verkkosivujen tekemiseen käytetty kolmas kulmakiviteknologia HTML:n ja JavaScriptin rinnalla, jota suurin osa verkkosivuista käyttää. [23]

CSS on suunniteltu pääsääntöisesti sen vuoksi, että dokumentista voidaan erotella esittämistapa ja sisältö toisistaan. Erottelulla saadaan aikaiseksi sisällön helppokäyttöisyyttä, taipuisuutta ja hallintaa, sillä yhden CSS-tiedoston kautta voidaan hallita useaa sivua yhtäaikaisesti ja saada helpommin yhtenevä ulkoasu sivuille.

Projektissa käytimme Garden-kirjastoa, joka on Clojurelle suunniteltu CSS-kääntäjä. Se muistuttaa hieman syntaksiltaan Hiccupia, se käyttää vektoreita esittämään määritykset. Käytännössä CSS-elementit kirjoitetaan Clojure-tyylillä ja kääntäjä kääntää koodit CSS-koodiksi. Tällä saavutetaan se hyöty, että tyylittelyn puolella voidaan ajaa loogiikkaa Clojurea käyttäen. [24]

```
[[:body
          { :padding-left "12em"
          :font-family "Times"
          :color "red"
          :background-color "blue"}]
[:ul.baari {
  :padding 0
  :margin 0
  :position "absolute"
  :top "3em"
  :left "2em"
  :width "8em"}
 [:li {
  :background "white"
  :margin "1em 0"
  :padding "1em"
  :border-right "2em solid black" }]
 [:a {
  :text-decoration "none" }
  [:.link {
  :color "blue" }]
  [:.visited {
  :color "purple" }]]]
[:h1 {
  :font-family "Helvetica" }]
[:address {
  :margin-top "2em";
  :padding-top "2em" }]
```

Esimerkkikoodi 9. Garden-koodia.

Esimerkkikoodi 9 on Clojure-koodi, joka on tehty Garden-kirjastoa käyttäen. Siinä on tehty sama CSS-luokka kuin liitteen 1 CSS-koodissa.

3.10 React-ohjelmointikehys

React on avoimen lähdekoodin JavaScript-kirjasto, jota käytetään rakentamaan käyttöliittymiä. Sen on alun perin luonut Facebook ja sitä ylläpitää nykyään myös Facebookin

lisäksi aktiivinen kehittäjäyhteisö. Ensimmäinen versio Reactista julkaistiin maaliskuussa 2013 ja sen loi Jordan Walke. [25; 26.]

React mahdollistaa suurien verkkosovellusten luonnin, jossa data voi muuttua ilman, että sivua tarvitsee ladata uudelleen. React:n päätavoitteet koostuvat seuraavista asioista:

- järjestelmän rakentaminen komponenteista
- mahdollisimman vähän uusia ominaisuuksia
- vakaus
- yhteensopivuus
- aikataulutus
- kehittäjä helppous
- oma sisäinen käyttö. [27]

Todennäköisesti näiden kulmakivien ansiosta React oli vuoden 2016 suurin nouseva teknologia. [28]

Projektissa halusimme käyttää nykyajan tehokkaimpia työkaluja ja tästä syystä päädyimme valitsemaan React-sovelluskehyksen.

3.11 Reagent-ohjelmointikehys

Reagent on minimalistinen rajapinta ClojureScriptin ja Reactin välillä. Sen avulla voi määritellä React-komponentteja käyttämällä ClojureScriptin omia funktioita ja dataa. Reagentin komponenttisyntaksi on samanlainen kuin Hiccup-kirjaston. [29]

Clojure Atom on Clojureen suunniteltu tapa hallita järjestelmän jaettua, synkronista ja itsenäistä tilaa. [30]

Reagentissa sovelluksessa tilan hallintaan käytetään Reagentin omaa atomia. Tästä syystä ulkoisten kirjastojen, kuten Reduxin käyttäminen ei ole tarpeellista. Kun Reagentin atom muuttaa tilaansa, se osaa muuttaa kaikkia elementtejä, joissa atom arvo on käytössä. [31]

Kuva 4. Esimerkkikoodin 10 tuottama sivu

```
(ns kauppasovellus.app
  (:require [reagent.core :as reagent :refer [atom]]))

(defonce ajastin (reagent/atom (js/Date.)))

(defn kello []
  (let [kello-str (first (clojure.string/split (.toTimeString
@ajastin) " "))]
    [:div.kello
     {:style {:color "#f34"}}
     kello-str]))

(defn nappi []
  [:button.nappi {:on-click #(reset! ajastin (js/Date.))}
   "päivitä kello"])

(defn komponentti []
  [:div
   [:h3 "Tervetuloa ostelemaan, kello on"]
   [kello]
   [nappi]])
```

```
(defn init []
  (reagent/render-component [komponentti]
                            (.getElementById js/document
"container")))
```

Esimerkkikoodi 10. Yksinkertainen Reagent esimerkki

Esimerkkikoodi 10 on toteutettu yksinkertainen verkkosivu Reagentin avulla. Se tulostaa tekstin ”Tervetuloa verkkokauppaan, kello on”, kellonajan ja napin verkkosivulle. Esimerkin suoritus alkaa init-funktiosta, jossa ensimmäisenä kutsutaan Reagentin render-component-funktiota ja toisena argumenttina annetaan vektori, jonka sisällä kutsutaan komponentti funktiota. Kolmas argumentti init-funktiossa on tavallisen JavaScript-metodin käyttämistä. Siinä haetaan sivun HTML-elementeistä se elementti, jolla id on container.

Toinen argumentti, joka kutsui komponenttifunktiota, palauttaa vektorin, joka sisältää HTML-elementtejä. Komponentti funktion viimeinen vektori taas kutsui toisia funktiota, joiden nimet olivat kello ja nappi.

Kello funktiossa määritellään paikallinen muuttuja let-komennolla ja annetaan sille nimi ”kello-str”. Arvon asettaminen ”(-> @ajastin .toTimeString (clojure.string/split " ") first)”-komennossa nuolifunktio tarkoittaa sitä, että ensimmäinen argumentti sijoitetaan seuraavan argumentin toiseksi parametriksi ja sen tulos sijoitetaan taas seuraavan argumentin toiseksi parametriksi, eli @ajastin atom -arvomuuttuja annetaan .toTimeString-funktiolle. Tämän jälkeen sen tulos annetaan split-funktion ensimmäiseksi argumentiksi ja viimeisenä split-funktion tulos asetetaan first-funktion ensimmäiseksi argumentiksi. Tämän hyöty on, että komennot saadaan loogiseen järjestykseen lukijalle. Komennon voi kirjoittaa myös muotoon ”(first (clojure.string/split (.toTimeString @ajastin) " "))”, jolloin se tekisi saman asian mutta lukeminen pitäisi aloittaa sisimmistä sulkeista.

Seuraavalla rivillä luodaan uusi elementti, jolle asetetaan luokkamuuttuja-elementin-luokka. Elementien luokkien avulla pystytään asettamaan elementeille eri tyylejä. Elementeille pystyy myös suoraan antamaan tyylimäärityksiä aaltosulkeilla ja niiden sisään asetetuilla tyylikomennoilla. Viimeisenä asetetaan kentän tekstiksi paikallisessa muuttujassa oleva kellonaika.

Kello-funktion jälkeen kutsutaan vielä nappi-funktiota, joka luo html-napin ja asettaa siihen on-click-kuuntelijan. Kun nappia painetaan, ajetaan kuuntelijassa oleva funktio. Tässä tapauksessa napin painaminen ajaa funktion, joka alustaa uudelleen kellonajan.

Kun kellonaika muuttuu, päivittyy myös html-elementti, jossa kellon arvoa on käytetty uuteen aikaan.

Reagent oli selkeä valinta projektin verkkosivujen toteuttamiseksi, sillä sen avulla pystyimme käyttämään React-komponentteja ja luomaan verkkosivut ja niiden toiminnallisuudet helposti.

3.12 ClojureScript-testaus

Projektin kehittämisessä käytimme aiemmin mainittua BDD-menetelmää, joka tarkoitti sitä, että kaikki tekemämme koodi täytyi testata. Tähän käytimme ClojureScriptin omia testauskirjastoja.

```
(scenario "ajan päivitys"
          (given "käyttäjä on sivulla" (system))
          (when "klikkaa nappia" (sim/click (sel1 (:container
container) [:#nappi]) nil))
          (then "hän näkee ajan päivittyvän" [container]
                (is (re-find #(str (first (clojure.string/split
(.toTimeString @ajastin) " ")))
                             (.-innerHTML (sel1 (:container con-
tainer) [:#kello])))))))
```

Esimerkkikoodi 11. ClojureScript-testiesimerkki

Esimerkkikoodissa 11 luodaan testitapaus, jossa käyttäjä olisi saapunut sivulle, simuloidaan napin painallus ja tarkastetaan, että aika on muuttunut. Tämänkaltaisia testejä loimme projektin kehityksessä.

3.13 Toteuttamamme projektin rakenne

Projektin front end -koodin pohjan loimme käyttämällä tenzing-nimistä työkalua. Tenzing rakentaa kansiorakenteen ja luo alustavan build.boot-tiedoston, johon se sisällyttää valmiiksi komennot paikallista kehitystä varten. Se myös mahdollistaa eri ohjelmistokehyksien lisäyksen projektiin muutamalla parametrilla luontikomennossa.

```
lein new tenzing kauppasovellus +reagent +garden +test
```

Esimerkkikoodi 12. Lein-komento, projektin luomista varten

Esimerkkikoodi 12:n komennolla saimme luotua projektin pohjan, johon sisältyi mukaan Reagent, Garden ja testit.

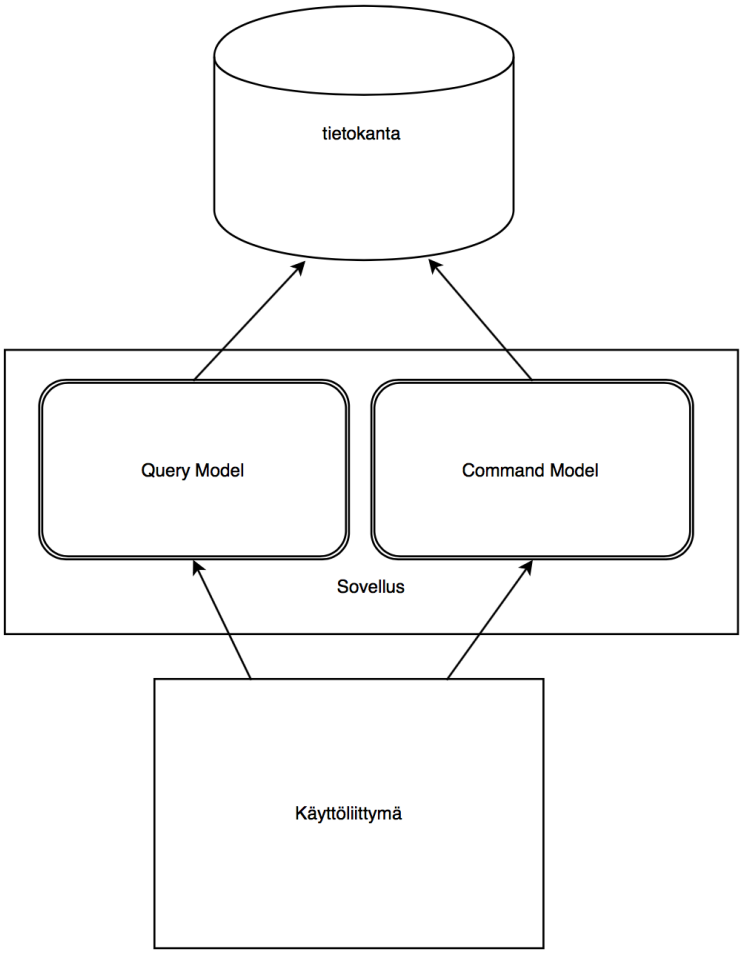
Kuva 5. Kuva luodusta sovellusrakenteesta

Kuvassa 5 näkyy sovelluksen rakenne, jonka pohjalta aloimme työstää projektin verkkosivuja. Resources-kansion alle tulee staattinen HTML-sivu, joka sisältää JavaScript-koodin lataamisen ja elementin, johon React-komponentit sidotaan kiinni. src-kansion alle tulevat projektin koodit, jossa on kansio Clojurlle ja ClojureScriptille. Clojure-kansioon tulevat Garden-tyylitiedostot. ClojureScript-kansioon tulee kaikki toiminnallinen koodi. test-kansiossa puolestaan luodaan BDD-testit. Boot-tiedostot tulevat suoraan projektin juureen.

Tämä toteutettiin siinä vaiheessa projektia, kun oli valittuna halutut teknologiat ja projektin alustavat UI-suunnitelmat olivat valmiina. Tämän jälkeen alkoi sovelluksen kehittäminen BDD-sovelluskehitysmallin tavoin. Ensimmäisenä kehitettävänä ominaisuutena projektiin koodattiin yksinkertaisin käyttäjätarina, jonka olimme alussa määritelleet.

3.14 CQRS arkkitehtuurina

CQRS muodostuu sanoista Command Query Responsibility Segregation ja sillä tarkoitetaan sovellusarkkitehtuurin rakennetta, jossa komento- ja kyselymallit on eroteltuna toisistaan. Komentomalli lähettää tiedon tietokantaan, ja kyselymalli hakee sieltä tietoa. Osassa tilanteista näiden erottelu on arvokasta, mutta väärintoteutettuna ne voivat aiheuttaa turhaa kompleksisuutta.



Kuva 6. CQRS-arkkitehtuurikuva

Yleisempi toteutus on, että tietokantaa käsittelee vain yksi malliosio, mutta projektissa päädyttiin käyttämään CQRS-menetelmää sen tuottamien etujen vuoksi. Kyseinen rakenne on hyödyllinen tilanteissa, joissa tietoa tallennetaan ja luetaan eri sijainneista. Toinen hyöty oli järjestelmän skaalautuvuuden vuoksi. Skaalautuvuuden saimme toteutettua Azure-funktioilla ja serverless-arkkitehtuurilla. [32]

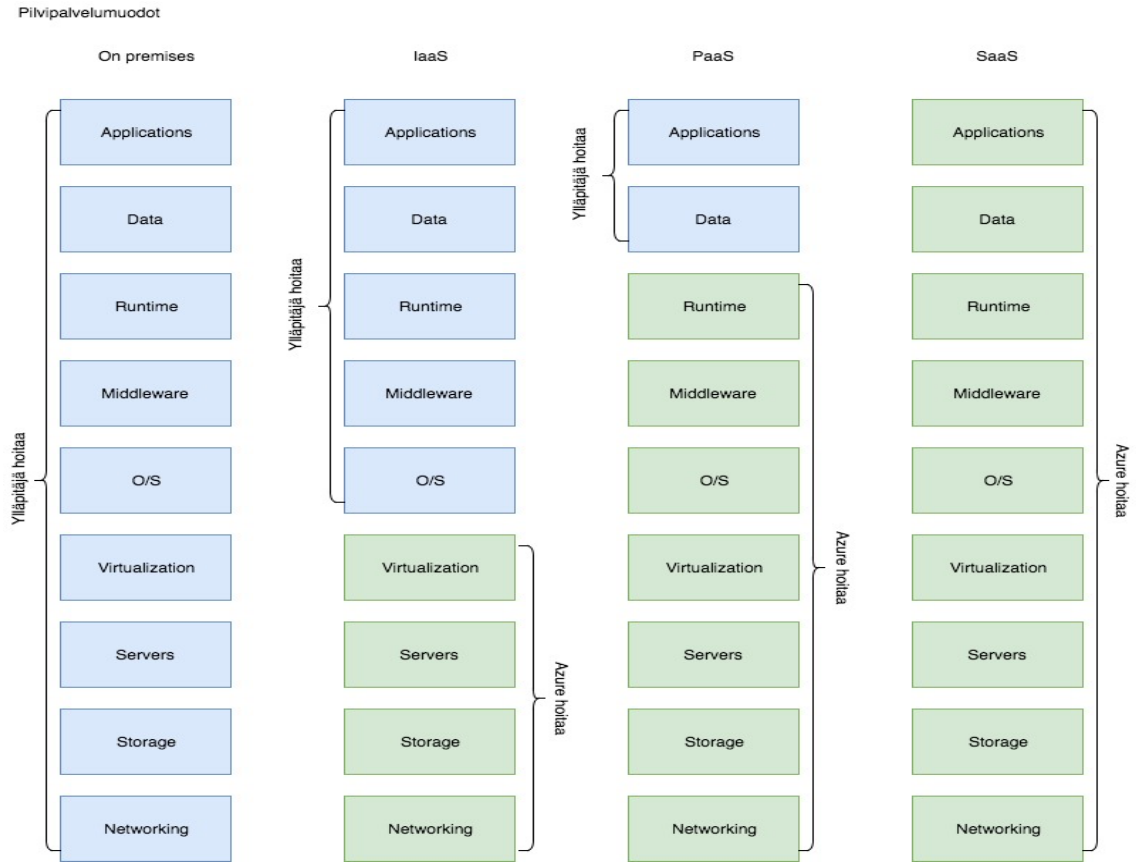
4 Azuren pilvipalvelu Backendinä

Tässä luvussa käsitellään Azuren pilvipalvelua ja sitä, mihin se on kykenevä. Samalla syvennytään arkkitehtuuriratkaisuihin, joita käytimme tämän projektin tekemiseen ja kuinka päädyimme näihin valintoihin, sekä perehdytään teknologioihin, joita nämä palvelut käyttävät.

4.1 Azure-pilvipalvelu

Microsoft Azure on Microsoftin kehittämä julkinen pilvipalvelu. Pilvipalvelu tarkoittaa laajaa kattausta resursseja, joita palveluntarjoaja antaa asiakkailleen käyttöön internetin välityksellä. Azure julkistettiin lokakuussa 2008 ja se julkaistiin helmikuussa 2010 nimellä Windows Azure. Nimi vaihdettiin Microsoft Azureksi 2013.

Pilvipalveluiden ominaispiirteitä on palveluiden itse provisiointi sekä joustavuus. Tällä tarkoitetaan, että asiakkaat voivat luoda palveluita helposti tarpeen vaatiessa, sekä sulkea niitä sitä mukaa, kun ne tulevat tarpeettomiksi. Azuressa myös laskutus menee käytön mukaan eikä siten, että palveluista joutuisi maksamaan etukäteen. Hallinnan näkökulmasta pilvipohjaisissa ratkaisuissa asiakkaat saavat pääsyn sovelluksiin, tietokantaan ja infrastruktuurisiin osiin ilman, että heidän täytyy itse ylläpitää ja päivittää niitä. [33]



Kuva 7. Azure-pilvipalveluiden tuotevastuut

Pilvipalvelut jaotellaan yleensä kolmeen eri palvelumuotoon, joita ovat software as a service (SaaS), infrastructure as a service (IaaS) ja platform as a service (PaaS). Kuva 7 kertoo visuaalisesti, kenen vastuulla minkäkin osan ylläpito on eri palvelumuodoissa.

4.1.1 SaaS-pilvipalvelumuoto

SaaS on palvelumalli, jossa valmista sovellusta pääsee käyttämään internetin välityksellä. Tällaisia sovelluksia on esimerkiksi Microsoft Office 365, jossa sovellus suoritetaan palvelimella eikä käyttäjän laitteella. [34]

SaaS-mallin etuina on se, että käyttäjä maksaa sovelluksesta vain sen mitä käyttää. Toinen etu on, että sovelluksen ylläpito on täysin palveluntarjoajan vastuulla. SaaS-sovellusten käyttö myös usein tukee useata sovellusalustaa, koska sovellus suoritetaan palvelimella eikä laitteessa. Tällöin käyttäjät voivat ottaa sovellukseen yhteyden omalta tietokoneeltaan, kännykästään tai tabletista, internetin välityksellä. [35]

Projektissa päädyimme käyttämään muutamaa eri SaaS-palvelumallin sovellusta. Näitä olivat muun muassa DocumentDB-tietokanta ja Servicebus. Käsittelemme näitä myöhemmässä luvussa.

4.1.2 PaaS-pilvipalvelumuoto

PaaS-malli tarkoittaa, että palveluntarjoaja tarjoaa alustan, jolla asiakkaat voivat kehittää, suorittaa ja hallita sovelluksia ilman monimutkaisia infrastruktuurisäädöksiä. Tällä saavutetaan samat hyödyt kuin SaaS-mallilla, mutta sovellus on toteutettava itse. [36]

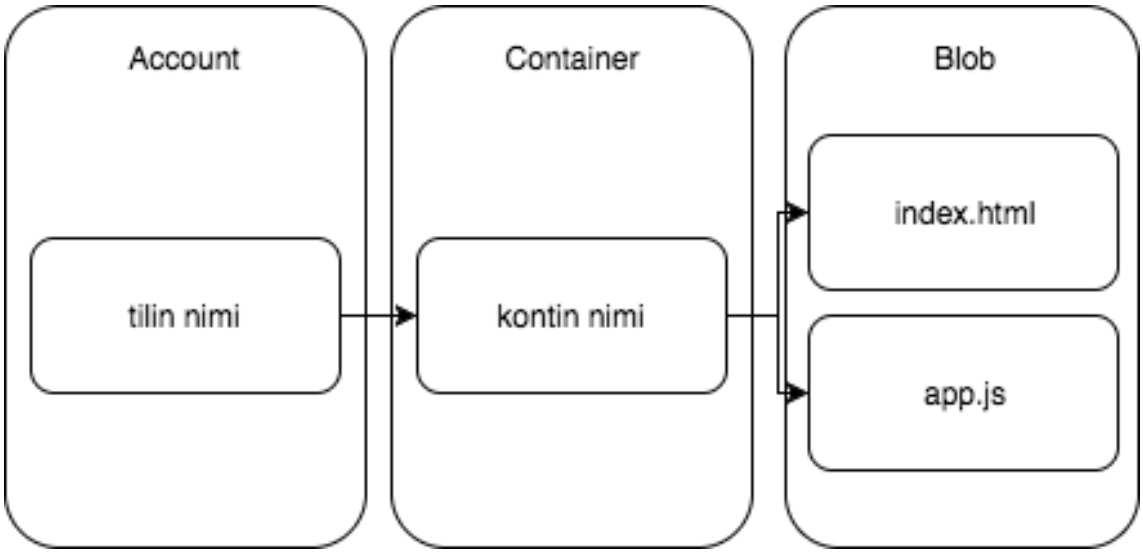
4.1.3 IaaS-pilvipalvelumuoto

IaaS-mallissa palveluntarjoaja antaa asiakkaalle käyttöön palvelimen pilvessä, joka luodaan tekemällä uusi virtuaalitietokone. Asiakkaalla on tässä mallissa täysi hallinta koko virtuaalikoneesta. Azure-virtuaalikonella pystytään toteuttamaan IaaS-ratkaisu. Projektin jatkuva integraatio toteutettiin käyttämällä juuri IaaS-ratkaisua. [36]

4.2 Blob storage -varastointipalvelu

Azure blob storage on varastointipalvelu, jonne tallennetaan suuria määriä rakenteetonta dataa, kuten tekstiä tai binääridataa. Niihin pääsee käsiksi http- tai https-yhteydellä. Blob storagen datan voi määrittää julkiseksi tai säilöä yksityiseen käyttöön.

Blob-palvelu koostuu seuraavista komponenteista account, container ja blob. Account tarkoittaa tiliä, jolle blob storage otetaan käyttöön. Blob storage voi sisältää useamman kontin, joiden alle data eli blobit tallennetaan. Blob taas tarkoittaa minkälaista tahansa tiedostoa. [37]



Kuva 8. Blob storagen komponentit

Blob storage oli sopiva ratkaisu meidän tarpeelle, sillä kun halusimme esittää sivuston edistystä asiakkaalle, pystyimme helposti lataamaan koodit Blob storageen ja näyttämään saavutetut tulokset. Toinen vaihtoehto olisi ollut perustaa Azureen oma verkkosovellus, mutta sen tekeminen olisi ollut työläämpää, sekä ylläpito ja käyttäminen kalliimpaa. Blob storage -palveluun pystyi helposti lisäämään aluksi upload-komennolla koodit, ja sivut tulivat heti näkyviin, kun blobin url-osoitteeseen otti yhteyttä selaimella. Kun projekti eteni ja saimme toteutettua jatkuvan integraation, rakensimme myös deploy-komennon, joka siirsi front end -koodit kontin alle automaattisesti haluttuun ympäristöön. Blob storage osoittautui tässä vaiheessa sopivaksi sijainniksi tallentaa front end -koodit pysyvästi sinne. Ratkaisu osottautui niin toimivaksi, että sitä voisi käyttää myös lopullisessa tuotteessa.

4.3 Azure-virtuaalitietokone Dockerin kanssa

Azure virtuaalitietokone on yksi Azuren tarjoamista heti saatavista, skaalautuvista tietojenkäsittelyresursseista. Virtuaalikone koostuu asiakkaan valitsemasta käyttöjärjestelmästä sekä halutusta käsittelytehosta. Tällä ratkaisulla asiakas välttyy fyysisen laitteiston ylläpitotyöltä. Virtuaalikoneet toimivat pystyttämisen jälkeen Azure-pilvessä. Tätä palvelumallia kutsutaan IaaS-ratkaisuksi. [38]

Azure tarjoaa myös valmiiksi asennettun ja konfiguroidun virtuaalikoneen Docker-virtuaalikoneiden käyttöä varten. Virtuaalikoneessa voi tämän jälkeen suorittaa haluamiaan Docker-kontteja. [39]

Jatkuvaa integraatiota varten loimme Docker-kontin, jota suoritetaan virtuaalikoneella. Tätä varten Azuren virtuaalitietokone osoittautui täydelliseksi ratkaisuksi Meidän ei tarvinnut ylläpitää tai hankkia erikseen tietokonetta. Pystytimme virtuaalikoneen Azure-pilveen sen jälkeen, kun Blob storage oli saatu toimimaan.

4.4 Palvelimeton arkkitehtuuri

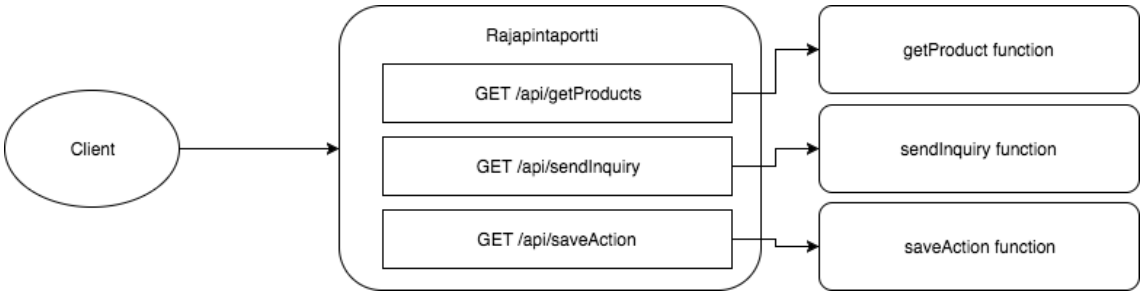
Palvelimeton arkkitehtuuri eli Function as a Service(FaaS) tarkoittaa sovellusta, joka on riippuvainen kolmannen osapuolen palveluista tai koodista, joka ajetaan lyhytaikaisissa konteissa. Luvussa keskitytään FaaS-ratkaisuun, sillä sitä käytettiin tämän projektin toteutukseen.

Tässä arkkitehtuurimallissa siirretään mahdollisimman paljon sovelluksen toiminnallisuudesta front end -toteutukseen ja omaa palvelinta ei tarvita järjestelmää varten. Tällä tavoin voidaan vähentää kustannuksia, sillä palveluista laskutetaan vain käytön mukaan.

Tarkemmin katsottuna FaaS-arkkitehtuurissa koodi suoritetaan tilattomissa ohjelmansuoritus-konteissa, jotka käynnistyvät tietyistä tapahtumista. Koodit ovat lyhyitä ja nopeita ja kontteja ylläpitää kolmas osapuoli.

FaaS-arkkitehtuurissa suurimmat edut ovat, että käyttäjä ei tarvitse ylläpitää omaa palvelinta. Sovellus ei ole sidonnainen mihinkään ohjelmointikieleen tai kehykseen, sillä funktiot tukevat useaa eri ohjelmointikieltä. Horisontaalinen skaalautuminen on täysin automaattista, joustavaa ja palvelun ylläpitäjä huolehtii siitä.

FaaS-mallissa on kuitenkin suuri rajoite, sillä funktioilla ei ole mitään yhteistä tilaa, vaan se on riippuvainen kaikesta datasta, mitä se saa laukaisussa itselleen. Tämän vuoksi FaaS-mallia kutsutaan tilattomaksi. Vaikka FaaS-malli on luonnostaan tilaton eli puhdas funktio, niin funktioihin voi silti yhdistää tilan käyttämällä esimerkiksi tietokantoja.



Kuva 9. FaaS-palveluiden rajapintaportti

FaaS-palvelut yleensä sisältävät rajapintaportin. Rajapintaportti tarkoittaa http-palvelinta, joka reitittää osoitteet niihin haluttuihin funktioihin. Rajapintaportti ottaa vastaan myös http-pyyntöjen parametreja ja antaa ne funktion argumenteiksi. Se myös muuttaa funktion tuloksen http-vastaukseksi ja palauttaa sen kutsujalle.

Hyötyjä FaaS-arkkitehtuurimallissa on sen edullisuus, sillä funktioiden käytöstä laskutetaan vain se, miten paljon niitä suoritetaan. Toinen etu on automaattinen skaalaus eli samaa funktiota voi kutsua useita kertoja ja ne ajetaan omissa instansseissaan.

Haittapuoliakin FaaS-arkkitehtuurimallista löytyy. Näitä ovat muun muassa palveluntuottajan hallinta, palvelu nojaa täysin palveluntuottajan sovellusten toimivuuteen. Palveluissa ei myöskään ole palvelimen sisäistä tilaa. [40]

Kun olimme perehtyneet FaaS-arkkitehtuurimalliin, totesimme sen olevan sopiva ratkaisu meidän projektin pilvitoteutukseen. Se olisi edullinen ja skaalautuva toteutus projektia varten. Jos palvelu ei olisi kannattava ja asiakas haluaisi lopettaa kehityksen, ei palvelusta aiheutuneiden kulujen tappio olisi niin suuri. Jos palvelunkehitystä taas jatketaan, olisi skaalautuvuus hyvä pohja jatkaa järjestelmän kehittämistä.

4.5 Azure Function

Azure Functions on Azuren toteutus FaaS-mallin pienten koodifunktioiden ajamiseen. Azure Functionssin avainominaisuudet ovat:

- useat eri ohjelmointikielet joista valita (C#, F#, Node.js, Python, PHP ja bash)

- laskutus vain käytön mukaan
- omien riippuvuuksien lisäys (järjestelmään voi lisätä omia haluamiaan kirjastoja)
- integroitu turvallisuus
- avoin lähdekoodi.

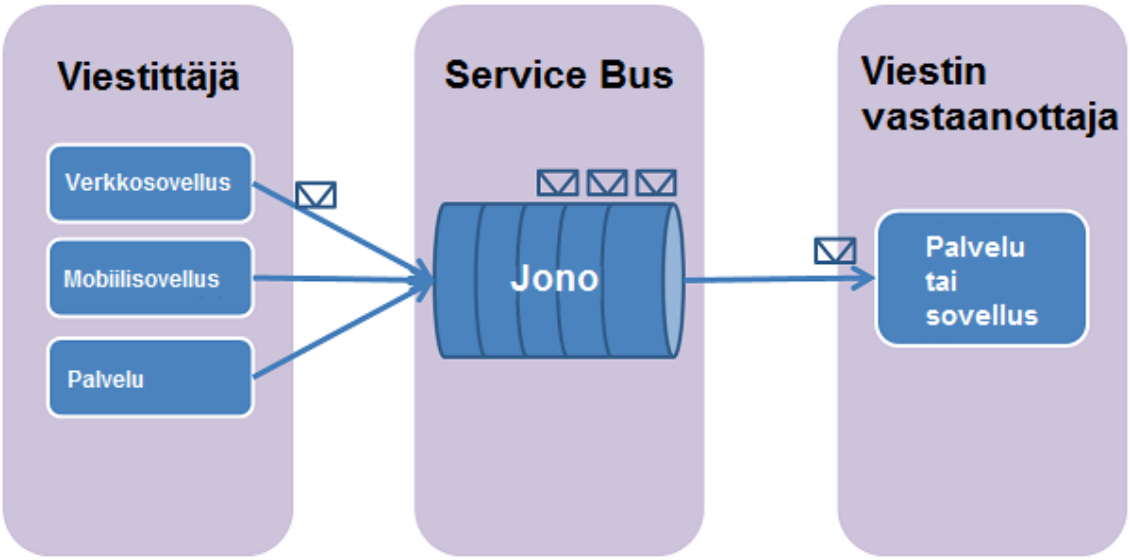
Azure.funktioiden käynnistys on helppo toteuttaa rajapintapalveluun tai mihin lähes tahansa muuhun Azure-pilven palvelun tapahtumaan. [41]

Projektin funktiot toteutimme Azure-funktioita käyttäen. Aloitimme funktioiden funktiosta, jota kutsuttaessa se hakee kategoriat ja tuotteet tietokannasta ja palauttaa ne. Ensimmäisessä versiossa palautimme staattista dataa, kunnes saimme tietokannat ja tuotetiedot. Tämän jälkeen muokkasimme funktiota palauttamaan oikeat tiedot. Toinen funktio, jonka loimme, oli tilauksen jättäminen ja tietojen tallentaminen. Tämän jälkeen MVP-toteutus oli valmiina pilven osalta.

4.6 Service Bus ja jonot

Service Bus on Azuren tarjoama kommunikointipalvelu. Service Bus:illa voidaan mahdollistaa eri pilvipalveluiden kommunikointi toisten palveluiden kanssa, niin sisäisten kuin ulkoisten. Service Bus on PaaS-palvelu, eli sen käyttöönotto tapahtuu helposti ja nopeasti. Service Bus tukee SOAP- ja REST-kutsuja ja viestitys voidaan toteuttaa yksisuuntaisena, pyyntö/vastaus- tai peer-to-peer-ratkaisuna.

Tyypillinen Service Busin toteutus on jonoratkaisu. Tämä ratkaisu käyttää välittäjäviestitystä, eli lähettäjän ei tarvitse saada yhteyttä vastaanottojaan heti, vaan viesti voidaan säilyttää jonossa, kunnes vastaanottavaan osapuoleen saadaan yhteys. Näin voidaan mahdollistaa luotettavuus ja eheys tiedonsiirrossa. Viestejä ja tietoa ei menetetä, vaikka jokin palvelu olisi pois päältä. Jonot käyttävät FIFO-periaatetta, eli viimeisin viesti prosessoidaan ensimmäisenä.



Kuva 10. Viestitys Service Bus -jonon kautta

Kuvassa 10 on esitetty ratkaisu, jossa viestittäjinä toimivat verkkosovellus, mobiilisovellus ja palvelu. Kaikki viestittäjät lähettävät viestinsä jonoon, josta viestit lähetetään eteenpäin muille palveluille tai sovelluksille.

Meidän ratkaisussa käytimme Service Busin jono -ratkaisua. Kun verkkosivulla on täytetty tilauslomake ja tehty tilaus, lähetetään tilaus front endistä Azuressa sijaitsevaan Service Busin jonoon. Tilaus lähetettiin REST-mallia käyttäen. Kun viesti oli jonossa, pystyi Azuren funktio poimimaan viestin jonossa ja prosessoimaan tilauksen eteenpäin. [42;43;44.]

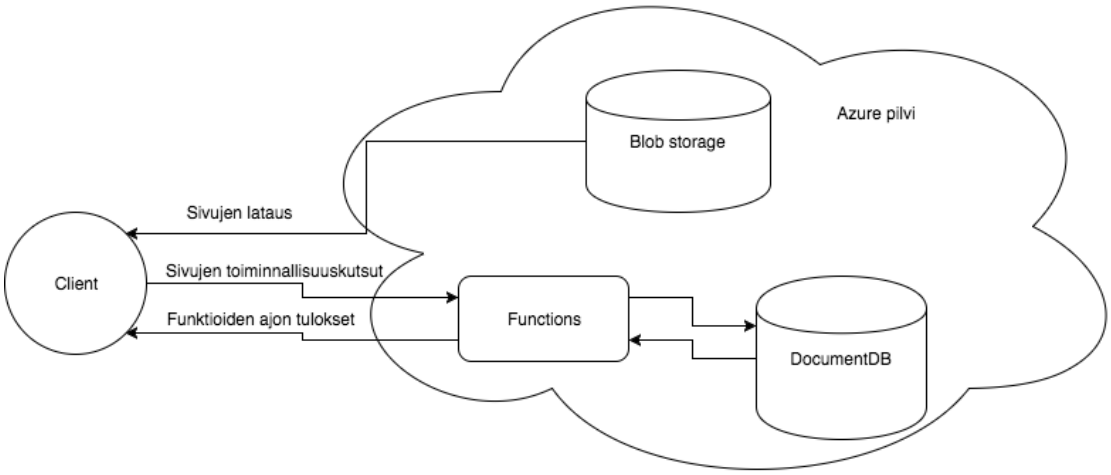
4.7 DocumentDatabase-tietokanta

DocumentDatabase on täysin hallittu NoSQL-tietokantapalvelu Azuressa. NoSQL tulee sanoista Not only SQL ja sillä tarkoitetaan perinteisestä relaatiotietokannasta eroavia tietokantoja. Tämä tarkoittaa sitä, ettei tallennetun tiedon tarvitse esimerkiksi noudattaa tiettyjä kenttiä, vaan se voi sisältää mitä tahansa dataa. NoSQL-malli kehitettiin, koska tavalliset relaatiotietokannat eivät olleet tarpeeksi tehokkaita, skaalautuvia ja joustavia. Tallennetut tiedot voivat olla minkä tahansa tyyppisiä binääri-olioita (esimerkiksi teksti, video tai JSON). NoSQL-tietokannoista haetaan dataa SQL-kyselyillä. DocumentDB tukee ainoastaan JSON-mallista tiedon tallennusta. Siinä on myös automaattinen indeksointi. [45;46.]

Projektin tietokannaksi valitsimme DocumentDB:n sen nopeuden vuoksi, sillä se pystyy tekemään monta samanaikaista hakua tietokantaan nopeasti. Se ei pakota tiettyjä kenttiä tietokantaan, mikä oli meille sopiva ratkaisu, koska emme vielä tässä vaiheessa pystyneet tietämään kaikkia kenttiä, joita tuotetietoihin tulee. Näitä tietoja olisi sitten helppo lisätä tai muuttamaan myöhemmin, kun tulee tarvetta.

4.8 Pilvi yhteenveto/arkkitehtuuri/kokonaiskuva

Aiemmissa kappaleissa esitellyistä Azure-palveluista saimme muodostettua meidän järjestelmän backend-toteutuksen käyttämällä FaaS-arkkitehtuuria.



Kuva 11. Kokonaiskuva pilviarkkitehtuurista

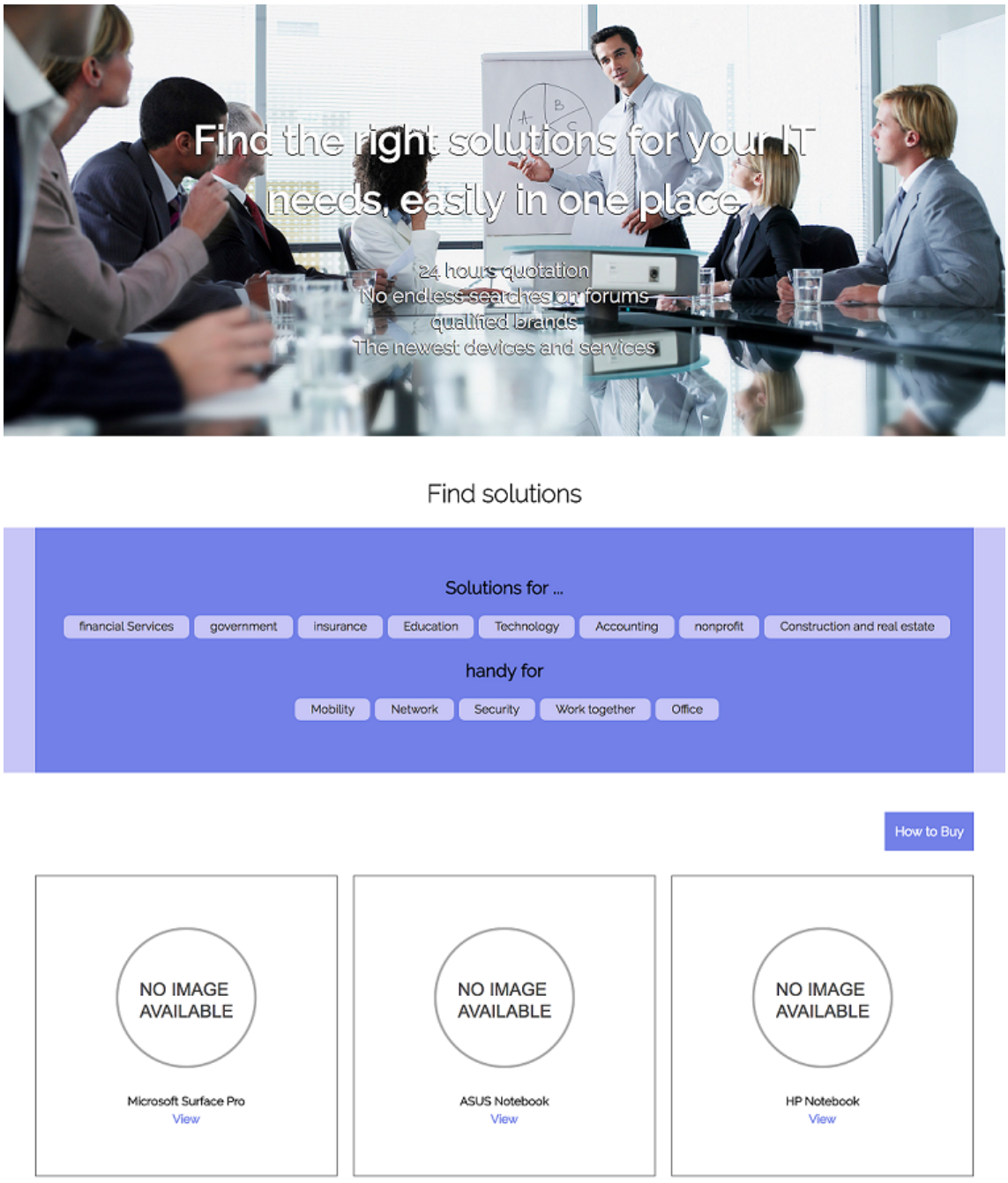
Kuvassa 11 näkyy toteutuksen kokonaiskuva. Ensimmäisenä käyttäjät saapuvat sivulle ja sivut latautuvat Blob storagesta. Tämän jälkeen tuotteet ladataan funktioiden kautta DocumentDB:stä. Kun tiedot palautuvat funktiolta, ne tulevat näkyviin selaimeen. Tämän jälkeen käyttäjä voi tehdä tuotetilauksia sivuilta, jotka tekevät uusia kutsuja funktioihin ja tallentavat tiedot.

5 Projektin yhteenveto

Viimeisessä luvussa käymme läpi viimeisen toteutuksen ja pohdimme käytettyjä teknologioita.

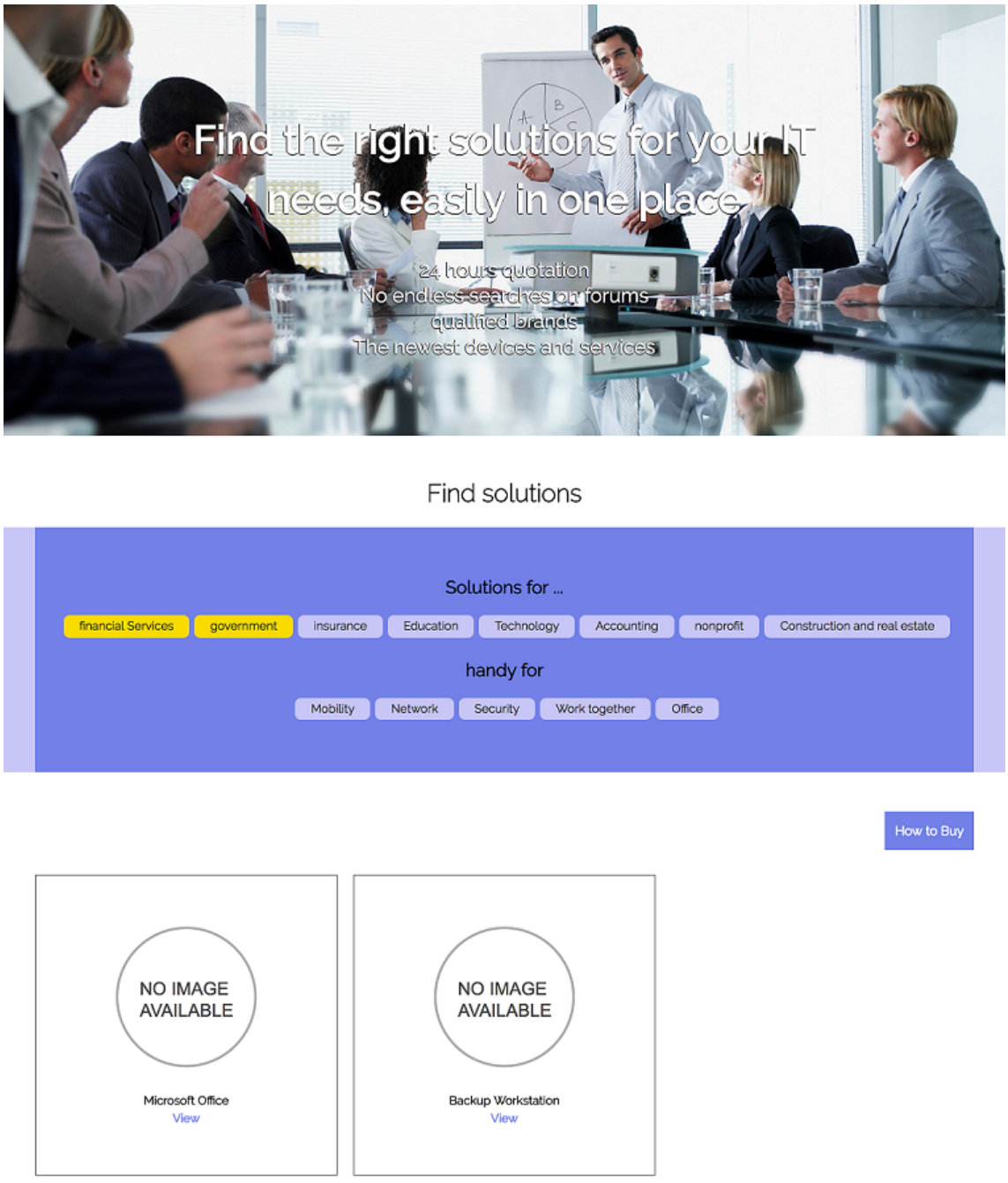
5.1 Pohdintaa teknologioista ja viimeinen toteutus tuotteesta

Clojure ja ClojureScript osoittautui erittäin sopiviksi ohjelmointikieliksi tähän projektiin, ja olimme tyytyväisiä niiden selkeyteen ja suorityskykyyn. Boot-työkalu osoittautui myös yllättävän monipuoliseksi ja tehokkaaksi niin kääntämiseen kuin myös testien ajamiseen. Jatkuvan integraation tärkeys korostui projektissa, sillä uusien ohjelmointikielten ja työkalujen kanssa tuli väistämättä virheitä koodiin. Jatkuvan intergroinnin avulla pystyimme löytämään ne nopeasti ja etenemään projektissa tehokkaasti. Eniten ongelmia meillä oli eri palveluiden kanssa Azuren pilvipalvelussa. Azure Functionssin kanssa huomasimme, että funktioilla kesti hetki käynnistyä, jos ne olivat olleet käyttämättä, mikä johti pieneen viiveeseen verkkosivua selatessa. Huomasimme myös, että Azuren palveluiden kehittämiseen olisi erittäin suotavaa olla käytössä Windows käyttöjärjestelmä -kehitysympäristössä ja siinä Microsoftin kehittämä ohjelmointiympäristö Visual Studio, sillä konfiguraatioiden muuttaminen Azuren verkkokäyttöliittymässä oli joskus epäluotettavaa ja hidasta.



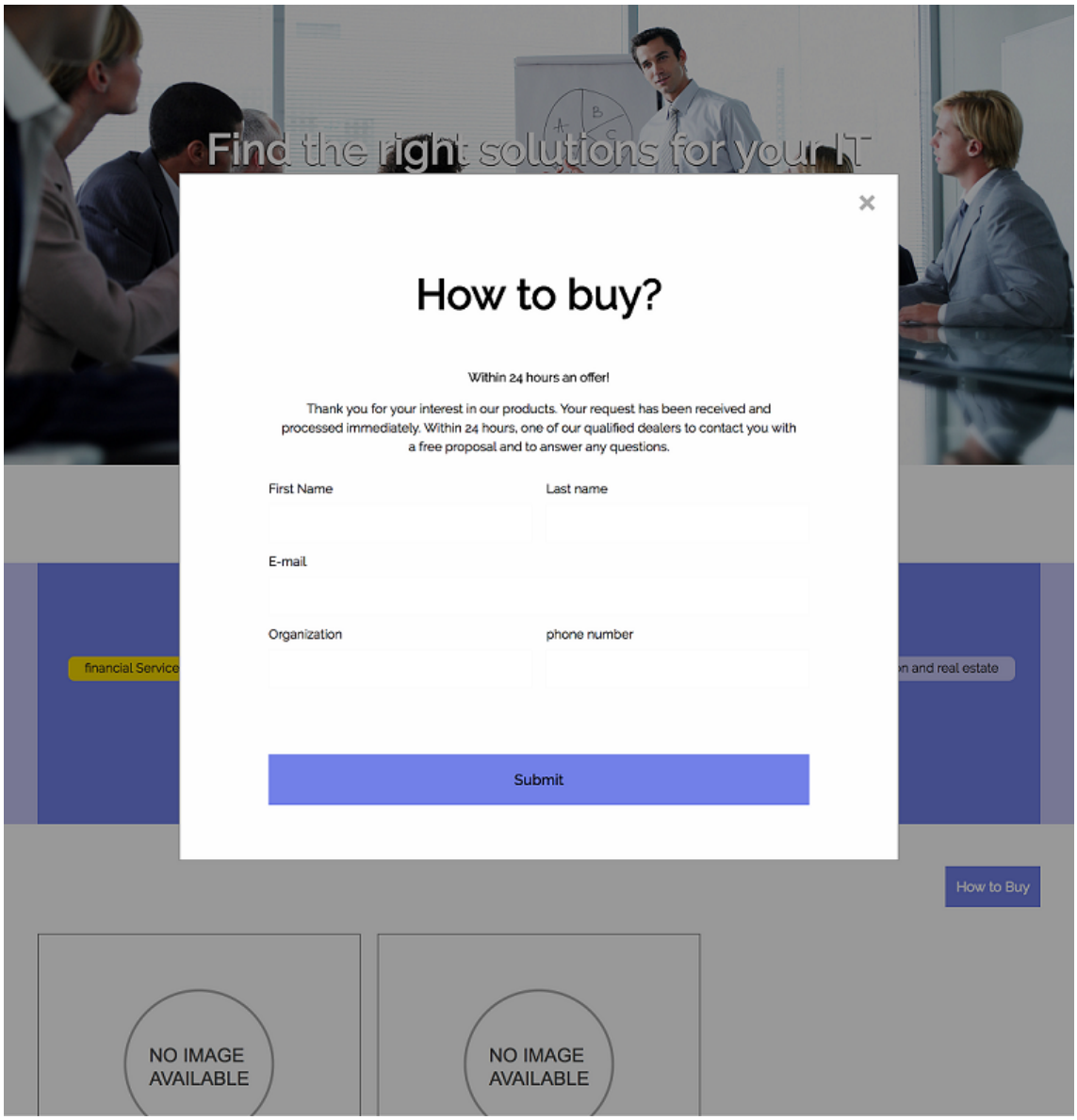
Kuva 12. Kuvankaappaus verkkokaupan etusivulta

Kuvassa 12 kuvankaappaus viimeisen toteutuksen etusivulta, jossa näkyy mahdollisuus tuotteiden rajaukseen tai yleisen tarjouspyynnön jättämiseen.



Kuva 13. Kuvankaappaus kategorioiden rajauksella

Kuvassa 13 tuotteita on rajatttu kategorian mukaan. Tässä vaiheessa voi joko mennä yksittäisen tuotteen tuotesivulla tai jättää tilaus näkyvistä tuotteista.



Kuva 14. Kuvankaappaus tilauslomakkeesta

Kuvassa 14 on kuvankaappaus tilauslomakkeesta tuotteista, jotka halutaan tilata.

Kuva 15. Kuvankaappaus tuotesivusta

Kuvassa 15 on kuvankaappaus tuotesivusta. Tällä sivulla on mahdollista pyytää tarjous-pyyntö yksittäisestä tuotteesta.

5.2 Yhteenveto

Tässä insinöörityössä kävimme läpi asiakasprojektin toteutusta projektin alusta loppuun kronologisessa järjestyksessä. Aloitimme projektin käsittelyn meille asetetuista alkuvaatimuksista, joita olivat tiukka aikataulu, MVP-toteutus ja Azure-pilvipalvelun käyttö. Tästä siirryimme käsittelemään sovelluskehitysmenetelmää, joka sopi parhaiten tämänlaisen projektin toteutukseen. Tiukat aikataulut ja MVP-toteutus pakottivat meidät käyttämään Lean Startup -menetelmää, joka oli täydellinen nopean toteutuksen ja testauksen vuoksi, joita asiakas halusi. Samalla syvennyimme sovelluksen kehitysmetodolgiaan, eli BDD:hen, millä saadaan mahdollisimman virheetöntä koodia ja huomataan, jos koodi hajoaa jossain vaiheessa kehitystä.

Kun alun metodologiat olivat valittu, aloimme perehtymään projektin seuraavaan vaiheeseen, joka oli kehitystyökalujen valinnat. Ensimmäisenä kävimme läpi versionhallinnan, jonne koodit tallennettiin kehityksen aikana. Versionhallinta on olennainen osa jokaista ohjelmointiprojektia, jolla vältyttiin ongelmilta, kun projektia oli kuitenkin tekemässä useampi henkilö.

Tämän jälkeen käytiin CI läpi, sillä projektin jatkuva kehitys täytyi saada näkyviin myös asiakkaalle. Kun CI oli tehtynä, uudet muutokset tulivat heti näkyviin sivulle, jonka annoimme myös asiakkaalle nähtäväksi. Tällä tavoin pystyimme lisäämään luottamusta asiakkaalta, sillä he pääsivät näkemään projektin etenemisen reaaliajassa.

Seuraava olennainen vaihe oli ohjelmointikielen valinta. Insinöörityössä käsittelimme valittua kieltä, ja se sisälsi esimerkkejä vastaavasta koodista, jonka toteutimme oikeaan projektiin. Samalla kävimme läpi hieman front end -toteutuksen CQRS-rakennemallia, joka sopi FaaS-toteutuksen kanssa yhteen.

Viimeinen pääluku oli, kuinka rakensimme projektin backend-toteutuksen Azure-pilvipalveluun. Käsittelimme Azure-palvelut, jotka osoittautuivat sopiviksi ratkaisuiksi projektia varten, sekä sitä, kuinka hyödynsimme niitä.

Kaikkien näiden vaiheiden jälkeen olimme saaneet projektin MVP-toteutuksen valmiiksi. Toteutus annettiin asiakkaalle ylläpidettäväksi tämän jälkeen, ja asiakas suorittaa Lean startup -menetelmän testausvaiheen. Asiakas oli tyytyväinen MVP-toteutukseen, mutta

projektin jatkuminen riippuu myös siitä, lisääkö se myyntiä. Projektin ensimmäinen vaihe oli kuitenkin onnistunut, ja molemmat osapuolet olivat tyytyväisiä.

Lähteet

1 Ries, Eric. 2011. The Lean Startup. Crown Publishing Group.

2 Minimum Viable Product(MVP). Verkkodokumentti. Techopedia. <https://www.techopedia.com/definition/27809/minimum-viable-product-mvp> Luettu 13.2.2017.

3 Wake, Bill. INVEST in Good Stories, and Smart Tasks. 2003. Verkkodokumentti. <http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>
Luettu 14.3.2017.

4 Käyttäytymislähtöinen kehitys(BDD). Verkkodokumentti. Dan North & Associates <https://dannorth.net/introducing-bdd/>
Luettu 14.3.2017.

5 Versionhallinta ja Git. Verkkodokumentti. Atlassian. <https://www.atlassian.com/git/tutorials/what-is-git>
Luettu 2.3.2017.

6 Docker ja konttiteknologia. Verkkodokumentti. Docker. <https://docs.docker.com/engine/reference/builder/>
Luettu 4.4.2017.

7 Continuous Integration. Verkkodokumentti. ThoughtWorks. <https://www.thoughtworks.com/continuous-integration> Luettu 29.4.2017.

8 Gitlab Runner. Verkkodokumentti. GitLab. <https://docs.gitlab.com/runner/> Luettu 29.4.2017.

9 John McCarthy. History of Lisp. 1996. Verkkodokumentti. <http://www-formal.stanford.edu/jmc/history/lisp/node6.html#SECTION00060000000000000000> Luettu 12.2.2017.

10 Fogus, Michael & Houser Chris. 2011. Joy of Clojure. Manning.

11 Tony Hey & Gyuri Pápay. 2014. The computing universe a journey through a revolution.

12 Building, Running, and the Repl. 2017. Verkkodokumentti. Daniel Higginbotham. <http://www.braveclojure.com/getting-started/#Using_the_REPL> Luettu 18.2.2017.

13 Changes to Clojure. Verkkodokumentti. Clojure. <https://github.com/clojure/clojure/blob/master/changes.md> Luettu 20.2.2017.

14 Closure compiler. 2016. Verkkodokumentti. Google. <https://developers.google.com/closure/compiler/> Luettu 14.4.17.

15 Sierra, Stuart. 2011. Introducing Clojurecript. Verkkodokumentti. Cognitect. <http://clojure.com/blog/2011/07/22/introducing-clojurescript.html> Luettu 15.8.17.

16 ClojureScript. Verkkodokumentti. <https://clojurescript.org/index> Luettu 26.2.2017.

17 Automated build. Verkkodokumentti. Agile Alliance <https://www.agilealliance.org/glossary/automated-build/> Luettu 26.2.2017.

18 Leiningen. Verkkodokumentti. <https://github.com/technomancy/leiningen> Luettu 10.3.2017.

19 Boot. Verkkodokumentti. <https://github.com/boot-clj/boot> Luettu 12.3.2017.

20 Why Boot is Relevant For The Clojure Ecosystem. 2014. Verkkodokumentti. <https://www.martinklepsch.org/posts/why-boot-is-relevant-for-the-clojure-ecosystem.html> Luettu 12.3.2017.

21 HTML5. 2014. Verkkodokumentti. W3C. <https://www.w3.org/TR/html5/> Luettu 16.3.2017.

22 CSS. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Learn/CSS> Luettu 17.3.2017.

23 Clark, Scott. Web based Mobile Apps of the Future Using HTML 5, CSS and JavaScript. Verkkodokumentti. <http://www.htmlgoodies.com/beyond/article.php/3893911/Web-based-Mobile-Apps-of-the-Future-Using-HTML-5-CSS-and-JavaScript.htm> Luettu 17.3.2017.

24 Garden. Verkkodokumentti. <https://github.com/noprompt/garden> Luettu 17.3.2017.

25 Fisher, Bill. How was the idea to develop React conceived and how many people worked on developing it and implementing it at Facebook? 2015. Verkkodokumentti. <https://www.quora.com/React-JS-Library/How-was-the-idea-to-develop-React-conceived-and-how-many-people-worked-on-developing-it-and-implementing-it-at-Facebook/answer/Bill-Fisher-17> Luettu 8.4.2017.

26 React. Verkkodokumentti. Facebook Inc. <https://facebook.github.io/react/> Luettu 8.4.2017.

27 Design Principles. Verkkodokumentti. Facebook Inc. <https://facebook.github.io/react/contributing/design-principles.html> Luettu 8.4.2017.

28 Develper Survey Results. 2016. Verkkodokumentti. Stackoverflow. <http://stackoverflow.com/insights/survey/2016> Luettu 8.4.2017.

29 Reagent: Minimalistic React or ClojureScript. Verkkodokumentti. <https://reagent-project.github.io/index.html> Luettu 8.4.2017.

30 Atoms. Verkkodokumentti. <https://clojure.org/reference/atoms> Luettu 15.4.2017.

31 Reagent. Verkkodokumentti. <https://github.com/reagent-project/reagent> Luettu 8.4.2017.

32 Fowler, Martin. CQRS. 2011. Verkkodokumentti. ThoughtWorks. <https://martinfowler.com/bliki/CQRS.html> Luettu 7.4.2017.

33 Cloud services. 2016. Verkkodokumentti. TechTarget. <http://searchcloudprovider.techtarget.com/definition/cloud-services> Luettu 22.4.2017.

34 Microsoft's Azure Cloud Platform Explained Part 1. 2014. Verkkodokumentti. Forbes. <https://www.forbes.com/sites/greatspeculations/2014/12/19/microsofts-azure-cloud-platform-explained-part-1> Luettu 23.4.2017.

35 Movahhed, Mandy. Why SaaS? 7 Benefits of Cloud vs. On-Premise Software. 2014. Verkkodokumentti. <https://www.handshake.com/blog/why-saas-cloud-benefits-vs-on-premise-software/> Luettu 23.4.2017.

36 Greiner, Robert. Windows Azure IaaS vs. PaaS vs. SaaS. 2014. Verkkodokumentti. <http://robertgreiner.com/2014/03/windows-azure-iaas-paas-saas-overview/> Luettu 23.4.2017.

37 Get started with Azure Blob storage using .NET. Verkkodokumentti. Microsoft Azure. <https://docs.microsoft.com/en-us/azure/storage/storage-dotnet-how-to-use-blobs> Luettu 23.4.2017.

38 Overview of Windows virtual machines in Azure. Verkkodokumentti. Microsoft Azure. <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/about> Luettu 26.4.2017.

39 Create Docker environment in Azure using the Docker VM extension. Verkkodokumentti. Microsoft Azure. <https://docs.microsoft.com/en-us/azure/virtual-machines/linux/dockerextension> Luettu 26.4.2017.

40 Martin Fowler. Serverless Architectures. Verkkodokumentti. <https://martinfowler.com/articles/serverless.html> 27.4.2017.

41 An introduction to Azure Functions. Verkkodokumentti. Microsoft Azure <https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview> Luettu 29.4.2017.

42 What is Event Hubs. Verkkodokumentti. Microsoft Azure. <https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-what-is-event-hubs> Luettu 29.4.2017.

43 Lock me down. Everything You Need to Know About Azure Service Bus Brokered Messaging (Part 1). Verkkodokumentti. <https://lockmedown.com/be-sure-with-azure-net-azure-service-bus-part-1/> Luettu 29.4.2017.

44 Service Bus messaging: flexible data delivery in the cloud. Verkkodokumentti. Microsoft Azure. <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview> Luettu 29.4.2017.

45 Introduction to Azure DocumentDB. Verkkodokumentti. Microsoft Azure. <https://docs.microsoft.com/en-us/azure/documentdb/documentdb-introduction> Luettu 29.4.2017.

46 Why NoSQL Database? Verkkodokumentti. Couchbase. <https://www.couchbase.com/nosql-resources/why-nosql> Luettu 29.4.2017.

CSS-esimerkki

```
body {
  padding-left: 12em;
  font-family: Times;
  color: red;
  background-color: blue }
ul.baari {
  padding: 0;
  margin: 0;
  position: absolute;
  top: 3em;
  left: 2em;
  width: 8em }
h1 {
  font-family: Helvetica }
ul.baari li {
  background: white;
  margin: 1em 0;
  padding: 1em;
  border-right: 2em solid black }
ul.baari a {
  text-decoration: none }
a:link {
  color: blue }
a:visited {
  color: purple }
address {
  margin-top: 2em;
  padding-top: 2em; }
```

Kuvio 1. Esimerkki CSS-koodista

Ylläolevassa koodiesimerkissä voi nähdä CSS-koodin syntaksin. Alussa määritellään mihin elementtiin tyylittelyt kohdistuvat, esimerkissä ensimmäisenä body. Sille annetaan eri tyylikomentoja aaltosulkujen sisällä ja jokainen eri komento erotellaan komennon nimellä, sitten kaksoispisteellä, valinnalla ja lopulta päätetään komento puolipisteeseen.