Sami Saariaho

PALVELUARKKITEHTUURIN SUUNNITTELU JA TOTEUTUS STARTUP-YRITYKSELLE

PALVELUARKKITEHTUURIN SUUNNITTELU JA TOTEUTUS STARTUP-YRITYKSELLE

Sami Saariaho
Opinnäytetyö
Syksy 2019
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä: Sami Saariaho
Opinnäytetyön nimi suomeksi: Palveluarkkitehtuurin suunnittelu ja toteutus startup-yritykselle
Opinnäytetyön nimi englanniksi: Designing and implementing system architecture for a startup
Työn ohjaajat: Sami Halonen (Fonl Oy) ja Veikko Tapaninen (OAMK)
Työn valmistumislukukausi ja -vuosi: syksy 2019
Sivumäärä: 79 + 1 liite

Tämän opinnäytetyön tilaaja on oululainen vuonna 2018 perustettu startup-yritys Fonl Oy (Fellows Of Northern Lights). Yrityksen ideana on toteuttaa monikanavainen asiakas-palveluntarjoaja-järjestelmä, jonka pääkanavana on mobiilisovellus. Opinnäytetyön kirjoituksen ajankohtana yrityksen tarkempi liikeidea ja järjestelmän toiminnallinen sisältö olivat salaisia, joten järjestelmää ja sen vaatimuksia kuvataan tässä opinnäytetyössä vain yleisesti. Työn tavoitteena oli suunnitella ja toteuttaa startup-yrityksen ja sen ideoiman järjestelmän kokonaisvaatimukset huomioon ottaen järjestelmäarkkitehtuuri ja sen pääkomponentit, kuitenkin rajattuna järjestelmän palvelinosuuteen (backend).

Järjestelmän ideointi ja vaatimusten määrittelyä aloitettiin syksyllä 2018. Järjestelmän toiminnallinen kuvaaminen suoritettiin yhdessä Fonl-tiimin kanssa kuvaamalla käyttötapauksia sekä loppukäyttäjille että muille järjestelmän toimijoille. Arkkitehtuurin ja ratkaisujen suunnittelua, teknologioiden valintaa ja toteutusta tehtiin Proof-of-concept-tyylisesti. Järjestelmän arkkitehtuuri ja tuotekehitysvaiheen ympäristö rakennettiin ja toteutustyö käynnistettiin vuoden 2019 alussa.

Työn tuloksena syntyi suunnitelma järjestelmän arkkitehtuurista sekä kehitysvaiheen ympäristö joka vastaa mahdollisimman paljon tuotantoympäristöä, lukuun ottamatta suorituskyky- ja saavutettavuusvaatimuksia. Tehdyt ratkaisut on todettu käytännössä toimiviksi ja vaatimusten mukaisiksi. Arkkitehtuurissa käytettiin hyödyksi kypsiä teknologioita, tuotteita ja kehyksiä (framework). Tämä mahdollisti kehitettävien toiminnallisuuksien rajaamisen selkeisiin rajattuihin komponentteihin, joiden kehittämistä voitiin tehdä tiimin kesken tehokkaasti.

Kehitysympäristön rakennusvaiheessa toteutettiin perustoiminnot ja tietoturva (salaus, autentikointi, auktorisointi) mutta tuotantoympäristön rakennusvaiheessa tulee vielä keskittyä toteuttamaan arkkitehtuurin mukaisten komponenttien saatavuus (high-availability) ja tietoturvaominaisuudet (ns. koventaminen).

Asiasanat: startup-yritykset, järjestelmäarkkitehtuuri, mikropalvelu, tapahtumapohjainen arkkitehtuuri, konttiteknologia, ArchiMate

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author: Sami Saariaho
Title of thesis: Designing and implementing service architecture for a startup
Supervisors: Sami Halonen (Fonl Oy) ja Veijo Tapaninen (OAMK)
Term and year when the thesis was submitted: autumn 2019
Pages: 79 + 1 appendice

This thesis was ordered by a startup-company Fonl Oy (Fellows of Northern Lights) which was founded in Oulu, Finland 2018. The business idea for the company is to implement multichannel customer/service-provider -system, for which the mobile application is the main usage channel. At the time of writing this thesis the company's precise business idea and functionality of the system were under non-disclosure agreement (NDA), so the system functionality and requirements are represented here only generally. The goal of this thesis was to do an architectural planning and implement the system's main components by taking into account the overall requirements for the system and delimiting this work to the backend of the system.

The composition of ideas and requirements for the system was started at autumn 2018. The description of functionality was conducted together with the Fonl-team by describing use cases for both end users and other actors of the system. Architectural planning, technological selection and implementation was done using Proof-of-concept-style of process. The architecture and development environment were constructed at the start of 2019 and the development of the system was started.

The outcome of this thesis was the plan for system architecture as well as the development environment which correlates as much as possible the production environment apart from the performance and accessibility requirements. Solutions chosen are proved to be useful and conformed to requirements. Mature technologies, products and frameworks were used in the architecture. This enabled the splitting the functionality into clear components thus making it possible to do the implementation efficiently in team.

During the build of the development environment main functionalities and security functions (encryption, authentication and authorization) were engineered but still, when building the production environment, one must focus on carrying out the missing features of high-availability and security hardenings.

Keywords: startup-companies, system architecture, microservice, event-driven architecture, containers, ArchiMate

ALKULAUSE

Haluan kiittää erittäin mielenkiintoisesta ja opettavasta työn aiheesta Fonl Oy:n perustajia Jari Utusta, Sauli Kippolaa sekä Sami Halosta, joka toimi myös yrityksen puolelta työni ohjaajana. Lisäksi haluan kiittää työn valvojaani Veikko Tapanista kannustuksesta, kärsivällisyydestä, ymmärryksestä sekä hyvistä neuvoista.

20.10.2019

Sami Saariaho

SISÄLLYS

SANASTO

API	Application Programming Interface Usein formaalisti dokumentoitu kuvaus teknisestä rajapinnasta, jonka kautta järjestelmän sisäistä tietoa voidaan hakea tai muuttaa. Tyypillisesti mikropalvelut toteuttavat omat integrointirajapintansa Apeina.
Backend	Palvelinalusta, palvelinpään toiminnallisuus, joka voi sisältää useita palvelimia eri tarkoituksiin. Tässä työssä käytetään termiä ”palvelualusta” kuvaamaan järjestelmän palvelulähtöistä arkkitehtuuria.
DevOps	Development and Operations, järjestelmäkehityksen toiminnan malli, jossa järjestelmän kehittäjien (dev) ja operaattoreiden (ops) roolit limittyvät tiimissä niin, että yksi tiimi pystyy toteuttamaan järjestelmän, saattamaan sen tuotantoon ja tukemaan sekä operoimaan sitä tuotannossa itsenäisesti.
Frontend	Käyttäjäsovellus, käytännössä usein järjestelmän käyttöliittymä.
Horisontaalinen ja vertikaalinen skaalautuvuus	Järjestelmän suorituskyvyn kasvattaminen joko laajentamalla toiminnallisuutta lisäämällä palvelimia (horisontaalinen) tai lisäämällä yhden palvelimen kapasiteettia (vertikaalinen).
Kyvykkyys	Järjestelmän toiminnallisiin vaatimuksiin kohdistuva looginen arkkitehtuurinen toiminnallisuus, joka toteutetaan valitun teknologian avulla.

Mikropalvelu	Järjestelmässä itsenäisesti toimiva, kooltaan pieni tai pienehkö komponentti, joka huolehtii järjestelmän yhdestä osa-alueesta. Osa-alue voi olla esimerkiksi liiketoiminnan tietokohde (domain) kuten Asiakas tai Tuote. Mikropalveluina toteutetun systeemin arkkitehtuuri on tyypiltään mikropalveluarkkitehtuuri. Mikropalvelu kapseloi osa-alueensa ja toteuttaa sen sisältämän logiikan ja datan käsittelyyn oman rajapinnan. Usein puhutaan API-rajapinnoista, mutta tämä ei ole ainoa toteutustapa.
MVP	Minimal Viable Product, pienin ja yksinkertaisin mahdollinen tuote, joka täyttää käyttäjien pääasialliset vaatimukset ja jolla yritys voi tulla ulos markkinoille.
Proof-of-concept	PoC. Nopeasti tehty yksinkertainen perustoteutus järjestelmän kyvykkyydestä, jolla pyritään osoittamaan toiminnon ja teknologian toimivuus vaatimukseen.
YAML	YAML (YAML Ain’t Markup Language tai Yet Another Markup Language) on selväkielinen merkintäkieli ja tiedostomuoto, jota käytetään usein konfiguraatiotiedostojen formaattina.

1 JOHDANTO

Opinnäytteen lähtökohtana on oululaisen vuonna 2018 perustetun startup-yrityksen Fonl Oy:n liikeidean toteuttamiseksi kehitettävä järjestelmä. Minulle tarjottiin syksyllä 2018 mahdollisuutta osallistua yrityksen järjestelmän suunnitteluun ja sen palvelualustan eli backend-osuuden kehittämiseen.

Työssä keskitytään käyttäjäsovellusten (frontend), käytännössä Android- ja iOs-mobiilisovellusten tueksi kehitettävään palvelualustaan, sen suunnitteluun ja toteutukseen kokonaisjärjestelmän vaatimusten näkökulmasta. Vaatimuksia tarkastellaan kohteena olevan järjestelmän ja startup-yrityksen erityispiirteiden näkökulmasta. Näitä startup-yrityksen tuotekehityksen erityispiirteitä ovat mm.

- tavoite saada mahdollisimman aikaisessa vaiheessa aikaiseksi tuote niillä kyvykkyyksillä ja toiminnallisuuksilla, joilla yritys voi tulla ulos markkinoille saadakseen nopeasti palautetta ja varmuutta kehitysvaiheessa tehtyjen oletusten pätevyydestä (ns. Mimimal Viable Product -tuote)
- tavoite epäonnistua nopeasti, jotta resurssit voidaan mahdollisimmat tehokkaasti kohdistaa mahdollisimman aikaisin oikeisiin kohteisiin
- yrityksen vision ja tuotekehitysvaiheessa olevan tavoitejärjestelmän määritysten kehittyminen ja jatkuva muuttuminen sekä näistä syntyvä tarve nopeasykliselle ja ketterälle sekä kokeilevalle Proof-Of-Concept-tyyppiselle kehittämiselle. (1.)

Tässä työssä esitellyt arkkitehtuuriset valinnat ja teknologia eivät suinkaan ole ainoat vaihtoehdot järjestelmän vaatimusten ratkaisemiseen. Etenkin kehitysvaiheessa kaikki ratkaisut eivät olisi välttämättömiä eikä niistä välttämättä saada irti kaikkia hyötyjä, joita ne mahdollistavat. Valitut ratkaisut ja teknologia on kuitenkin suunniteltu ja valittu järjestelmän myöhempää laajempaa käyttövaihetta ajatellen. Toiminnan laajentumista varten ratkaisuissa ja valinnoissa on kiinnitetty huomiota skaalautuvuuteen, korvattavuuteen ja siirrettävyyteen esimerkiksi toiselle pilvialustalle.

Toisaalta tässä työssä ei keskitytä ja syvennytä yritystason arkkitehtuurillisiin kehyksiin ja prosessimalleihin kuten TOGAF Enterprise Architecture Framework

(2), yrityksen sisäisiin kehittämisen standardeihin, arkkitehtuurin johtamiseen (governance) ja mittareihin.

Tämä opinnäytetyö on jaettu kuuteen lukuun. Luvussa 2 tehdään katsaus opinnäytetyön aiheena olevien järjestelmäarkkitehtuurin perusmallien historiaan. Tarkoituksen on antaa lukijalle perusymmärrys ominaisuuksista ja problematiikasta, jotka ovat johtaneet tässä työssä tutkittujen ja valittujen ratkaisujen syntymiseen.

Luvussa 3 esitellään teknologioita, joita voidaan hyödyntää suunniteltujen arkkitehtuurikomponenttien toteuttamisessa. Luvun tarkoitus on antaa lukijalle perustieto eri teknologioista ja niiden ominaispiirteistä.

Luku 4 esittelee tämän opinnäytetyön kohteena olevan järjestelmän ja järjestelmän piirteet, jotka asettavat vaatimuksia valittavalle arkkitehtuurille. Luvun esittelee piirteet perustuen seuraaviin johtaviin teemoihin:

- startup-yrityksen erityispiirteet
- monikanavaisuus
- monipuolinen toiminnallisuus
- tietoturvallisuus ja tietosuoja
- suorituskyky, skaalautuvuus
- tekninen alusta.

Luku 5 esittelee arkkitehtuuriset teknologiavalinnat pohjautuen luvun 4 vaatimuksiin ja esittää perustelut kullekin valinnalle.

Luku 6 esittelee suunnitelman mukaisen arkkitehtuurin käyttäen AchiMate-kuvauskieltä (3) ja -komponenttimallia, jossa arkkitehtuuri kuvataan kolmella abstraktiotasolla: taso 0 kuvaa järjestelmän kokonaisarkkitehtuurin ja ympäristön esittäen itse kohteena olevan järjestelmän vielä mustana laatikkona (black-box). Taso 1 avaa kohdejärjestelmän ja esittää sen loogiset pääkomponentit ja niiden väliset liitynnät (white-box). Taso 2 esittää pääkomponenttien sisäiset rakenteet ja yhteydet toisiinsa tai järjestelmän rajapintoihin. (4.)

Viimeisessä luvussa tehdään yhteenveto järjestelmästä, valituista arkkitehtuurikehyksistä ja ratkaisuista sekä pohditaan arkkitehtuurin haasteita, parannuskohtia ja jatkokehittämistä.

2 JOHDATUS ARKKITEHTUURIMALLEIHIN JA MALLINTAMISEEN

Tässä luvussa alustetaan ja kuvataan tämän opinnäytetyön kohteena olevan järjestelmän arkkitehtuurin perustana olevat mallit. Tarkoitus on kuvata lähtökohtia ja vaiheita, jotka ovat vaikuttaneet ja johtaneet kuvattujen teknologioiden kehitykseen.

Luvussa 2.6 esitellään myös arkkitehtuurin kuvaukseen käytetty ArchiMate-notaatio ja -kieli (3), jota käytetään myöhemmin kuvaamaan kohdejärjestelmän arkkitehtuuri (luku 6).

2.1 Client-server ja kerrosarkkitehtuurit

Hyvin yleinen arkkitehtuurinen malli tietojärjestelmäkehityksessä on ollut jo 1960-luvun keskuskoneiden aikakaudelta alkaen client-server-malli. Siinä järjestelmän toiminnallisuus on jaettu käyttäjän koneen toiminnallisuuteen sekä palvelintoiminnallisuuteen. Tässä niin sanotussa kerrosarkkitehtuurissa käyttäjän kone (client) ja palvelin (server) keskustelevat suoraan keskenään tietoverkon välityksellä. Ensimmäisiä keskustietokoneita käytettiin ”tyhmistä” suoraan palvelimeen liitetyistä terminaaleista, jotka sisälsivät pelkästään käyttäjän operoiman käyttöliittymän. Kaikki tieto prosessoitiin palvelimella, jota kutsuttiin keskuskoneeksi (mainframe). Vähitellen tekniikan kehittyessä ja tietokoneiden yleistyessä myös clienttien toiminnallisuus kehittyi. (5.)

Tähän kehitykseen johtivat silloisten tietoverkkojen pienet siirtokapasiteetit ja nopeudet, jolloin prosessointitehoa tarvittiin enemmän paikallisesti clientissa. Ensimmäiset tietoverkot perustuivat tiedostojen asynkroniseen jakamiseen eli tiedosto ladattiin ensin client-koneelle prosessoitavaksi ja lähetettiin takaisin keskuskoneelle muiden käyttäjien saataville. Verkkojen laajentuessa tällaisen tiedostopohjaisen tiedonjaon kapasiteetti tuli kuitenkin esteeksi ja kehiteltiin tekniikka, jossa tiedon keskitettyyn tallennukseen vaihdettiin clientia palveleva tiedostopalvelin tietokantapalvelimeksi ja tietoa palautettiin clientille vain pyydetty määrä kokonaisen tiedoston sijaan. Näin kehittyi nykyaikaiset relaatiotietokantajärjestel-

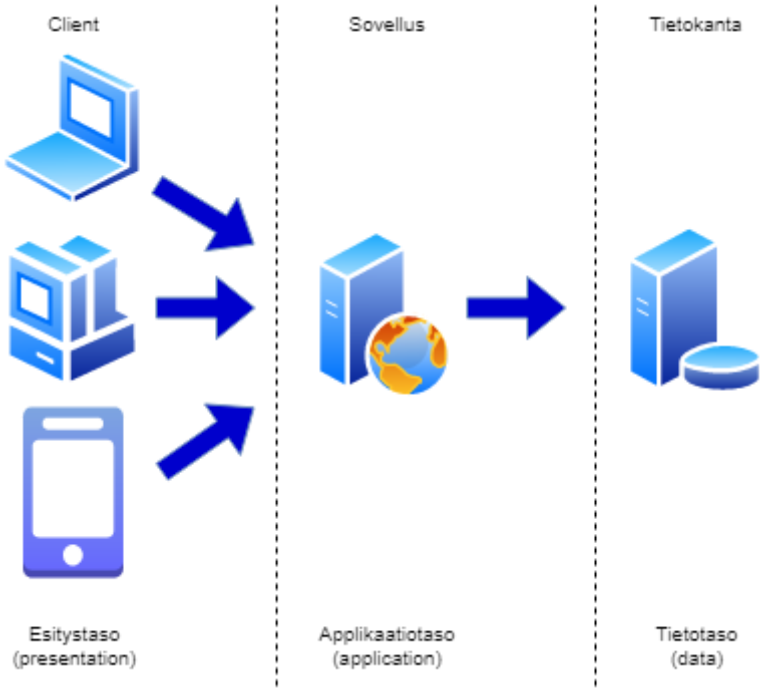
mät. Tämä johti tarvittavan verkon tiedonsiirtokapasiteetin määrän pienenemiseen. Toisaalta taas tämä malli kasvatti clientin ohjelmistoa ja eriytti käsittelylogiikkaa jokaiseen tietokantapalvelinta käyttävään clienttiin, jolloin ohjelmistojen kehitys ja ylläpito vaikeutui.

Merkittävin hyöty client-server-mallista on resurssien eriyttäminen sinne missä sitä on mahdollista helposti ja halvalla saada. Kevyemmät clientit vaativat vähemmän tilaa ja prosessoritehoa, kun taas raskas käsittelylogiikan ja datan käsittely tehdään isoissa palvelimissa, joissa on paljon tilaa ja muistia sekä prosessoritehoa.

Käyttäjäkoneen client-osuus sisältää kaksikerroksisessa client-server-mallissa vain käyttöliittymätoiminnallisuuden (sen mitä käyttäjä näkee). Client-toiminnallisuudesta kutsutaan varsinaista järjestelmän toiminnallisuutta palvelimelta (backend). Nykyaikaisia client-server-järjestelmiä ovat mm. sähköpostijärjestelmät ja www-sivustot. Myös nykyaikaiset puhelimet ja niiden mobiilisovellukset nähdään myös yhtenä käyttäjän käyttöliittymänä ja yhteyden muodostus serveriin on mahdollista mobiiliverkon kautta langattomasti.

Kaksikerroksisen client-server-arkkitehtuurin huono puoli on sen skaalautuvuus, sillä tallennettavan tiedon sisältämän palvelimen monistaminen palvelemaan kasvavaa käyttäjämäärää on kallista ja vaatii tiedon replikointia tietokantapalvelinten välillä mikä taas vaarantaa tiedon eheyden.

Client-server-arkkitehtuurin toinen tyyppi on kolmikerrosarkkitehtuuri, jossa clientin rooli on sama kuin edellä kuvatun kaksikerrosarkkitehtuurissa, mutta tiedon tallennus on eriytetty varsinaisesta clienttia palvelevasta ja käsittelylogiikkaa sisältävästä serveristä. Kolmikerrosarkkitehtuuri kuvataan kolmena loogisena kerroksena, joita ovat tietotaso (datalayer), sovellustaso (applicationlayer) ja esitystaso (presentationlayer) (6). Kuvassa 1 on esitetty kolmikerrosarkkitehtuurin tasot ja niiden väliset yhteydet.



KUVA 1. Kolmikerrosarkkitehtuurin tasot

Tänä päivänä puhutaan jo N-kerrosarkkitehtuureista, joissa clientin kutsujen käsittely on tieto- ja sovellustason lisäksi eriytetty lisätasoihin mm. skaalautumisen mahdollistamiseksi käyttäjämäärän kasvaessa. Yksi tällainen esimerkki on High-Availability-välityskoneiden (proxy) ja erillisten http-palvelinten käyttö sovellustason ja clientin välissä. (5.)

Kerrosarkkitehtuuria noudatetaan myös nykyaikaisena pidetyssä mikropalveluarkkitehtuurissa. Myös siinä tietotaso on yleensä eroteltu omaksi kerroksekseen tietokantana tai muuna tietovarastona. Arkkitehtuuri voi koostua eri tason mikropalvelutasoista (7) kuten myös omista proxy-koneista, kuormajakajista ja palvelurekistereistä (8).

2.2 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuurille ei ole kehitetty varsinaista virallista määritystä, mutta sen ilmentymistä on eritelty tiettyjä tunnusmerkkejä, jotka tuntuvat pätevän useampaan arkkitehtuuritoteutukseen. Näitä tunnusmerkkejä ovat seuraavat:

- Järjestelmä on toteutettu itsenäisinä ja korvattavissa olevina sekä asennettavina palvelukomponentteina. Palvelut toteuttavat dokumentoidut ja

määrämuotoiset rajapinnat, joiden kautta muut komponentit integroituvat palveluun.

- Järjestelmän kehittäminen on organisoitu liiketoiminnallisten kyvykkyyksien mukaan, ei kehittäjien osaamisalueiden mukaan. Sen sijaan, että eri osaajat esim. käyttöliittymä- ja tietokantatiimeissä hoitavat kehitystyötä omissa siiloissaan, koostuu kehitystiimi eri tyyppisistä osaajista ja siten tiimi voi hoitaa yhden kyvykkyysalueen palvelukokonaisuutta kokonaisuudessaan.
- Järjestelmän kehittämistä tehdään tuotteina, ei projekteina. Tuotetiimi omistaa palvelun ja vastaa siitä koko sen elinkaaren ajan aina määrittelystä tuotantoon ja ylläpitoon.
- Järjestelmän palveluintegraatiot on toteutettu ”tyhminä” putkina. Tällä tarkoitetaan sitä, että järjestelmän toiminnallisuutta ei hajauteta palveluiden ulkopuolelle ja niiden väliseen integraatioon. Esimerkkinä tällaisesta hajautetusta mallista on myöhemmin tässä luvussa esitelty ESB-välikerros. Jokainen palvelu omistaa ja hoitaa sen oman tietomallin ja liiketoimintalogiikan sekä vastaa sen saamaan palvelukutsuun itsenäisesti saamansa input-tiedon perusteella.
- Järjestelmän arkkitehtuurin johtaminen on hajautettua. Jokainen palvelukokonaisuus voi toteuttaa toiminnallisuutensa siihen parhaiten sopivilla teknologioilla ja työkaluilla. Ainoa asia, josta palvelun tulee sopia käyttäjiensä kanssa, on palvelun rajapinta.
- Järjestelmän tieto on hajautettu. Palvelut huolehtivat omista liiketoimintamalleistaan ja tiedoistaan, jotka usein on hajautettu omiin tietokantoihin. Hajautettu tietomalli tuo myös omat haasteensa, jota kuvataan myöhemmin tässä luvussa ja luvussa 2.3.
- Järjestelmän infrastruktuuri on autonomista ja automatisoitua. Palvelutiimit pyrkivät tehostamaan kehitystä mahdollisimman paljon automatisoidulla testauksella ja jatkuvalla integroinnilla ja asennuksella (Continuous Integration / Continuous Delivery).
- Järjestelmäarkkitehtuuri on suunniteltu myös virhetilanteiden varalle. Hajautettu palvelumalli on virheherkkää ja palveluiden katkoksiin ja virhetilanteisiin tulee varautua ja niistä tulee toipua mahdollisimman sulavasti,

jopa automaattisesti, aiheuttamatta järjestelmän toiminnalle ja käyttäjälle liikaa harmia. Palveluiden hallintaprosessit, monitorointi ja lokitietojen kerääminen on hoidettu kattavasti, jotta virhetilanteisiin voidaan reagoida ja niistä voidaan toipua tehokkaasti.

- Järjestelmän kehitys on inkrementaalista ja arkkitehtuuri on muutoksiin sopeutuvaa. Kehitystä voidaan tehdä ketterästi ja muuttuviin vaatimuksiin voidaan reagoida nopeasti. Muutokset voidaan hoitaa usein korvaamalla olemassa oleva komponentti (palvelu tai palvelun toiminnallisuus) toisella. (9.)

Vaikka mikropalveluista ja mikropalveluarkkitehtuurista on puhuttu aktiivisesti vasta 2000-luvun alkuvuosista alkaen, on sen ydinkonseptien kehittyminen ollut pitkäaikaisen evoluution tulos. Greg Young käy tätä evoluutiopolkua läpi omassa esityksessään ”The Long Sad History of Microservices” (10).

Young tuo esille esityksessään, että mikropalvelukonsepti ei ole mikään uusi juttu ja sen juuret ulottuvat aina 1960-luvun lopulle saakka. 1970-luvun alussa kehitetty olio-ohjelmointi (Smalltalk-ohjelmointikieli) ja sen perusajatus itsenäisestä ”oliosta”, pienestä itsenäisestä tietokoneesta on yksi perusta mikropalveluajattelulle. Toisaalta myös Actor-malli eli toimija, joka toimii ja suorittaa tehtävän itsenäisesti saamaansa tietoon perustuen sekä erilaisia ohjelmointiparadigmoja noudattavat ohjelmointikielet (kuten lokiikkapohjaiset Prolog, funktionaaliset Erlang ja Lisp) ovat lähtökohtia mikropalvelukonseptille. (10.)

Young näkee myös hajautettuun tiedonvälitykseen kehitetyt protokollat Microsoftin DCOMin (Distributed Component Object Model) ja CORBA:n (Common Object Request Broker Architecture) perustana mikropalveluiden hajautetun arkkitehtuurin ja palveluiden välisen tiedonsiirron lähtökohtana (10).

Isoiksi monoliiteiksi rakennetut järjestelmät vastaavat huonosti järjestelmän kehittämisessä ja ylläpitämisessä vaadittaviin ominaisuuksiin ja sen ongelmana nähdään mm. seuraavat:

- toteutuksen iso koko ja monimutkaisuus sekä siitä johtuva vaikeus ymmärtää toiminnallisuutta sekä korjata tai muuttaa sitä

- muutoksen vaikutusten arviointi on vaikeaa ja muutos vaatii aina massiivista koko järjestelmän testausta
- toiminnallisuuden päivitys vaatii aina koko järjestelmän uudelleen asennuksen
- järjestelmän skaalaaminen on vaikeaa
- huono vikasietoisuus (single-point-of-failure)
- yksittäisen toiminnon tai kokonaisen komponentin korjaus tai korvaus toisella, paremmin tarpeeseen sopivalla toiminnallisuudella tai teknologialla ei onnistu helposti. (11.)

Nämä järjestelmän kehittäjiä ja ylläpitäjiä turhauttavat puutteet johtivat osaltaan mikropalveluarkkitehtuurin kehittämiseen (9).

Internetin syntyminen ja 2000-luvun alussa yleistyneet alustariippumattomat protokollat kuten HTTP ja sen päälle rakennettu SOAP-protokolla XML-esityskielineen mahdollistivat järjestelmien väliset helpot synkroniset integraatiot. Tietoverkkojen laajentuminen ja verkkotekniikan kehittyminen mahdollisti toimintojen hajauttamisen jopa globaalisti. Näistä lähtökohdista syntyi nykyaikaisen mikropalveluarkkitehtuurin perusta, palvelukeskeinen arkkitehtuuri (Service Oriented Architecture, SOA). (12.)

SOA:n perusajatus on rakentaa järjestelmäarkkitehtuuri palveluiden ympärille yhden ison monoliittisen järjestelmän sijaan. Vaikka alkuperäisen SOA:n perusajatukset pätevät edelleen, sen toteutukset yleensä epäonnistuivat (13). Yksi SOA:n toteutusmalleista on toteuttaa erillinen ESB-palveluväylä (Enterprise Service Bus), siis välikerros, joka toimii reitittäjänä ydinjärjestelmän ja palveluiden väliselle liikenteelle. Yksi ESB:n ongelmista on se, että toimiessaan kaiken liikenteen solmukohtana sen rikkoutuminen, ylläpidon huoltotoimenpiteet ja virheet sen toiminnallisuudessa aiheuttavat ongelmia palveluiden väliselle liikenteelle ja estävät näin koko järjestelmän toiminnan (ns. singe-point-of-failure). Haasteena palveluväylässä on myös sen sanomanvälityslogiikan kuvaaminen ja toteuttaminen, joka vaatii erityisosaamista ja luo uuden ja ylimääräisen ohjelmoitavan, konfiguroitavan ja ylläpidettävän kerroksen järjestelmäarkkitehtuuriin. (14.)

Palveluväylien kehittymisen sijaan palvelukeskeinen arkkitehtuuri kehittyi ja yksinkertaistui erillisten toimintojen ja ohjelmistojen rajapintoihin (Application Programming Interface, API) mikä johti lopulta erillisen palveluväylän tarpeen vähenemiseen. Palveluväylän sijaan puhutaankin nykyään API-hallinta -tuotteista, joiden avulla API:t voidaan julkaista, suojata ja hallita kokonaisuutena yhdestä paikasta (15).

Nykyaikaiset mikropalvelut toimivat itsenäisesti ja tarjoavat palveluja suoraan muille, jopa kolmansien osapuolten järjestelmille avointen rajapintojen kautta. Monimutkaisemman SOAP-protokollan rinnalle ja nykyään jo lähes korvaajaksi syntynyt HTTP-protokollan päällä toimiva REST-protokolla (Representational State Transfer) on helpottanut myös käyttäjän ja palvelun välisiä suoria Point-To-Point-integraatioita (12).

Mikropalveluarkkitehtuurin yhtenä haasteena pidetään sen hajautettua tietokanta-arkkitehtuuria. Tyypillisesti mikropalveluarkkitehtuurissa yksi palvelu huolehtii ja kapseloi yhden liiketoimintakohteen tiedot ja toiminnallisuudet. (9.). Tällöin useisiin kohteisiin kohdistuvia transaktioita on vaikea hallita ja ne voivat pahimmillaan virhetilanteissa aiheuttaa tiedon eheysongelmia. Mikropalveluiden välisten transaktioiden ongelmaa voidaan ratkaista käyttäen Saga-mallia, jota kuvataan luvussa 2.3.

2.3 Saga-malli

Mikropalveluarkkitehtuuri toteutetaan usein Domain-Driven-mallin mukaisesti hajautettuna tietokantaratkaisuna, jossa jokainen mikropalvelu huolehtii ja kapseloi oman liiketoimintakohteen tiedot ja toiminnallisuuden (9).

Tällaisessa hajautetussa mallissa useaan kohteeseen kohdistuvien toiminnallisuuksien transaktionhallinta tulee ongelmaksi. Usean mikropalvelun välisten tietoa päivittävien transaktioiden tulisi huolehtia, että jokaisen palvelun hoitama kohde jää lopulta oikeaan tilaan. Mikäli esimerkiksi kahden mikropalvelun alueelle kohdistuvan transaktion toisen palvelun toiminta pysähtyy virheeseen, tulisi myös ensimmäisen palvelun tekemä toiminto peruuttaa. Tämän hallinta on haastavaa hajautetussa mikropalveluarkkitehtuurissa.

Tätä haastetta voidaan lähteä ratkaisemaan toteuttamalla ns. Saga-mallin mukainen arkkitehtuuri. Saga-mallissa hajautettujen systeemien välille toteutetaan erillinen hallintakomponentti, joka jakaa kokonaistransaktion erillisten systeemien omiksi pieniksi paikallisiksi transaktioiksi. Keskeinen hallintakomponentti välittää systeemien väliset tapahtumat ja ”orkestroi” siten kokonaistransaktion hallinnan systeemien välillä. (16.)

Esimerkkinä Saga-mallin käytöstä voidaan kuvata seuraava kuviteltu transaktio kahden mikropalvelun välillä tilanteessa, jossa käyttäjä tekee uuden tilauksen järjestelmään:

1. *Asiakas luo uuden tilauksen, jolloin tilaustransaktio käynnistyy.*
2. *Mikropalvelu 1 (tilauspalvelu) luo uuden tilauksen järjestelmään tilaan ”odottaa” (paikallinen transaktio) ja lähettää Saga-hallintakomponentille tiedon uuden tilauksen syntymisestä mukaan lukien tilauksen summan.*
3. *Saga-hallintakomponentti välittää tilaustiedot mikropalvelu 2:lle (tilipalvelu).*
4. *Mikropalvelu 2 veloittaa saamansa tilaustiedon summan asiakkaan tililtä (paikallinen transaktio).*
5. *Mikropalvelu 2 lähettää Saga-hallintakomponentille tiedon onnistuneesta tiliveloituksesta.*
6. *Saga-hallintakomponentti välittää tiedon onnistuneesta tiliveloituksesta mikropalvelu 1:lle.*
7. *Mikropalvelu 1 päivittää tilauksen tilan ”odottaa”-tilasta ”valmis”-tilaan (paikallinen transaktio).*
8. *Tilaustransaktio päättyy.*

Mikäli annetussa esimerkissä mikropalvelu 2:n paikallinen tiliveloitus-transaktio epäonnistuisi esimerkiksi tilin saldon riittämättömyyden takia, lähettäisi mikropalvelu 2 tiedon epäonnistuneesta tiliveloituksesta Saga-hallintakomponentin kautta mikropalvelu 1:lle, joka puolestaan voisi joko poistaa tilauksen tai päivittää sen esimerkiksi tilaan ”ei-veloitettu”.

Yksi vaihtoehto toteuttaa Saga-hallintakomponentti on tapahtumapohjainen arkkitehtuuri ja Event Sourcing -malli. Event Sourcing -mallissa järjestelmän tila muodostetaan järjestelmän toimintojen emittoimien tapahtumien avulla. Event Sourcing -mallin mukainen Saga-hallintakomponentti ottaa vastaan ja julkaisee

tapahtumia mikropalveluilta, jotka kuuntelevat hallintakomponentin kanavaa/kanavia ja reagoivat kanavaan saapuviin tapahtumiin tapahtuman sisältämien tietojen mukaisesti. (17.)

Tapahtumapohjaista arkkitehtuuria on kuvattu luvussa 2.5.

2.4 Imperatiivinen, deklaratiivinen ja reaktiivinen malli

Jotta ymmärretään paremmin seuraavassa luvussa esiteltävän tapahtumapohjaisen arkkitehtuurin ideaa, on syytä selventää kahta toisistaan poikkeavaa paradigmaa eli imperatiivista ja deklaratiivista mallia. Reaktiivinen malli on yksi deklaratiivisen mallin alatyyppi (18).

Imperatiivinen malli on perusta kaikille ohjelmointikielille (18). Se on malli, jossa ohjelman kulku etenee luonnollista ja järjestelmällistä polkua ja sitä on sen takia helppo kirjoittaa ja ymmärtää. Imperatiivinen malli keskittyy siihen, *miten* ohjelma suoritetaan ja määrittelee ohjelman toiminnan käskyinä, jotka suoritetaan synkronisesti ja jotka muuttavan ohjelman tilaa. (19.)

Deklaratiivinen malli taas keskittyy siihen, *mitä* suoritetaan. Deklaratiivinen malli määrittelee ohjelmiston logiikan, mutta ei sen suoritusjärjestystä. Suoritus on tyypiltään asynkronista ja koodia ei välttämättä pysty seuraamaan täysin lineaarisesti. (20.)

Reaktiivinen malli on deklaratiivisen mallin tyyppi, jossa ohjelman kulku perustuu tapahtumiin, tietovirtoihin ja tiedon muutoksiin reagoimiseen.

Esimerkki imperatiivisesta ohjelmakoodista on esitetty kuvassa 2. Siinä summa-muuttujan arvo ei muutu, vaikka summan toisen tekijän arvoa muutetaan, koska summa on jo laskettu ennen muutosta. Ohjelman kulkua ja toimintaa on helppo seurata ja lopputulos helppo ymmärtää.

```
1: int arvo1 = 20;
2: int arvo2 = 20;
3: int summa = arvo1 + arvo2;
4: print(summa)    // tulostaa luvun 40
5: arvo1 = 10;
6: print(summa)   // tulostaa edelleen luvun 40
```

KUVA 2. Esimerkki imperatiivisesta ohjelmasta

Sen sijaan kuvassa 3 on esitetty sama ohjelma, mutta sen toiminta on ajateltu deklaratiivisesti. Summa-muuttujan arvo lasketaan viimeisimmillä muuttujien arvo1 ja arvo2 arvoilla eli summa-muuttuja lasketaan uudelleen arvo1:n muuttumisen jälkeen. Ohjelman kulun ja toiminnan ymmärtämisen kannalta on tiedettävä, millä tavalla muuttujien arvo muuttuu ja lopputulos voidaan vasta sen jälkeen päätellä (tosin annettu esimerkki tämän suhteen on hyvin yksinkertaistettu).

```
1: int arvo1 = 20
2: int arvo2 = 20
3: int summa = arvo1 + arvo2
4: print(summa) // tulostaa luvun 40
5: arvo1 = 10
6: print(summa) // tulostaa luvun 30
```

KUVA 3. Esimerkki deklaratiivisestä ohjelmasta

Reaktiivisessa kuvan 4 esimerkissä arvo-taulukosta luodaan virta (stream), jossa taulukon numerot ”virtaavat” ykkösestä yhdeksään. Virtaa seurataan (observe) ja aina virran arvon muuttuessa se tulostetaan kaksinkertaisena. Esimerkki on jälleen triviaali, mutta ohjelman toiminnan ymmärtämiseksi olisi tiedettävä (jos edes mahdollista), millä tavalla tietovirta muuttuu ajan funktiona.

```
1: int[] arvot = [1,2,3,4,5,6,7,8,9];
2: arvot.stream.observe(n -> print(n*2)) // tulostaa 2,4,6,8,10,12,14,16,18
```

KUVA 4. Esimerkki reaktiivisestä ohjelmasta

Imperatiivista ja deklaratiivista sekä reaktiivista käsitettä voidaan laajentaa ohjelmoinnista ja ohjelmakoodista myös järjestelmän arkkitehtuuriin: onko arkkitehtuuri rakennettu komponenteista, jotka käyttävät toisiaan synkronisesti loogisessa selkeässä järjestyksessä, vai käytetäänkö asynkronista, tietovirtoihin perustuvaa ja tiedon muutoksiin reagoivaa tapahtumalähtöistä mallia. Luvussa 2.5 esitellään reaktiiviseen mallin perustuvaa tapahtumapohjaista arkkitehtuuria.

2.5 Tapahtumapohjainen arkkitehtuuri

Digitaalinen liiketoiminta on yhä enemmän tapahtumakeskeistä. Uudet digitaaliset ja digitalisaation myötä uudistettavat vanhat liiketoiminnat, kuten myös asiakkaat, verkostoituvat ja kommunikoivat yhtä monimutkaisemmin tuottaen koko ajan enemmän ja enemmän uudenlaista tietoa. Tieto pitää tehokkaasti jakaa ja ilmaantuvaan tietoon pitää tehokkaasti ja mukautuvasti reagoida. (21.)

Tämä trendi vaatii myös teknologialta enemmän ja enemmän reaaliaikaisuutta ja tapahtumakeskeisyyttä. Tähän vaatimukseen johtaa myös koko ajan kasvava joukko uusia digitaalisia palveluita ja laitteita (Internet of Things), jotka integroituvat toisiinsa muodostaen monimutkaisia verkkoja. Myös erilaisten päätelaitteiden määrä kasvaa sekä palveluiden sisältö monipuolistuu.

Liiketoiminnallinen ja teknologinen kenttä siis monimutkaistuu koko ajan ja perinteiset integraationmallit eivät enää sovellu nykytarpeisiin. Perinteinen imperatiivinen toteutusmalli, jossa tiedon tai tapahtuman tuottaja kutsuu olemassa olevia tietyn toiminnan toteuttavia toiminnallisuuksia (push-malli), ei mahdollista tarvittavaa toiminnan joustavuutta ja laajennettavuutta. Reaktiivinen toteutusmalli, jossa tuotettuun tietoon ja tapahtumiin voidaan reagoida tuottajan tiedostamatta, kuka tapahtumia kuluttaa ja millaisella toiminnallisuudella (publisher/subscriber- ja pull-malli), on sen sijaan joustava ja helpommin laajennettava malli. (22.)

Reaktiivinen malli voidaan toteuttaa tapahtumapohjaisella arkkitehtuurilla (Event-Driven Architecture, EDA). Tapahtumapohjainen arkkitehtuuri perustuu liiketoiminnassa tai sen ulkopuolella tapahtuvien merkittävien tapahtumien (event) tuottamiseen ja käsittelyyn. Tapahtuman käsittely tapahtuu täysin itsenäisessä, tapahtuman tuottajasta riippumattomassa manuaalisessa tai automatisoidussa toiminnoissa, tapahtuman käsittelijässä. Käsittelijöiden toiminnallisuutta ja määrää ei ole rajoitettu. Käsittelijä käynnistyy tapahtuman tulessa saataville ja suorittaa omat toiminnallisuutensa tapahtuman sisältämän tiedon perusteella. Toiminnallisuus voi olla yksinkertainen tapahtuman tiedon prosessointi ja tallennus, tai siitä voidaan käynnistää toinen palvelu tai tuottaa uusi tapahtuma. (23.)

Tapahtumavirta (event stream) on EDA:n yksi tapahtuman prosessointityyli. Siinä erityyppisiä tapahtumia tuotetaan jatkuvana virtana tapahtumavirtaan ja käsittelijät ”kuuntelevat” tapahtumia virrasta ja käsittelevät niitä heti tapahtumien saapuessa. Tapahtumavirtaa käytetään reaaliaikaisten toimintojen toteutukseen, jossa tapahtumat aiheuttavat erilaista toiminnallisuutta eri puolilla liiketoimintaa, sen ympäristöä tai liiketoimintajärjestelmää. (23.)

Tapahtumavirtatoteutusta voidaan käyttää myös hyödyksi mikropalveluarkkitehtuurin palveluiden välisten transaktioiden hallintaan, kuten luvussa 2.3 esiteltiin. Yksi teknologia tapahtumavirran toteuttamiseen esitetään luvussa 3.7.

2.6 ArchiMate-kuvauskieli

ArchiMate on avoin ja ilmainen yritystason arkkitehtuurin (Enterprise Architecture) visuaaliseen kuvaamiseen tarkoitettu kieli, joka standardisoi arkkitehtuurin kuvauksessa käytetyt analyysit ja notaatiot. ArchiMaten kuvauskieli on tarkoitettu yksinkertaistamaan laajojen yritysarkkitehtuurien kuvaamista ja ymmärtämistä. (3.)

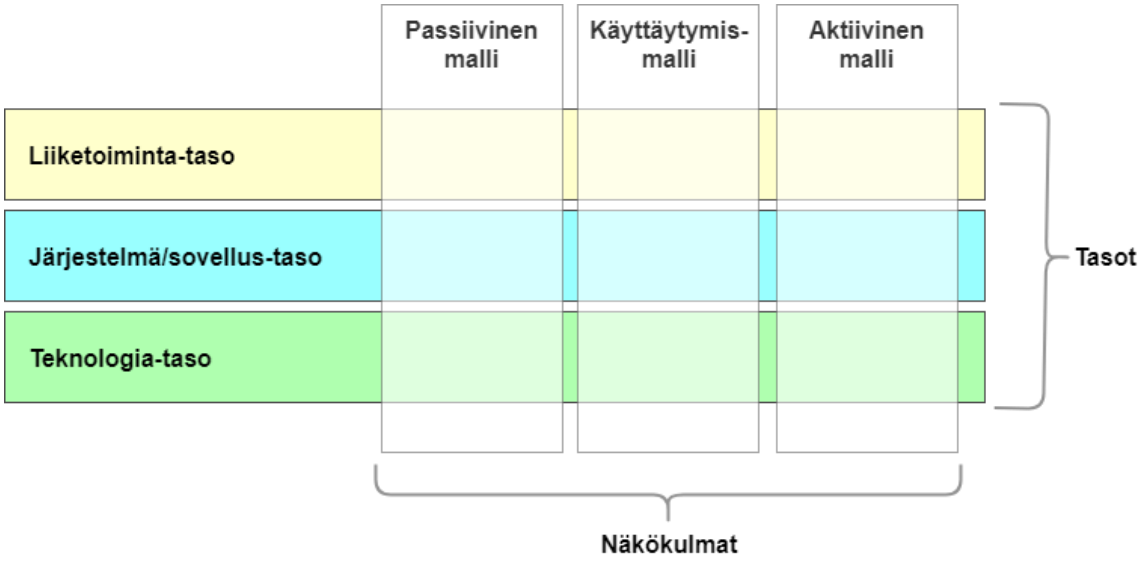
ArchiMaten viitekehys (Core Framework) määrittelee sen ydinkielen (Core Language) arkkitehtuuriset elementit. Viitekehyksen perusrakenne on esitetty kuvassa 5. Elementtejä voidaan tarkastella ydinkielen eri tasoilta, joita ArchiMatessa ovat:

- Liiketoiminta-taso (Business Layer); taso esittää liiketoiminnalliset, yrityksen asiakkaille lisäarvoa tuottavat palvelut, jotka suoritetaan liiketoimintaprosessien ja toimijoiden (actors) avulla
- Järjestelmä/Sovellus-taso (Application Layer); taso esittää liiketoimintaa tukevat järjestelmäpalvelut sekä järjestelmät, jotka toteuttavat kyseiset palvelut
- Teknologia-taso (Technology Layer); taso esittää teknologiset palvelut kuten tiedon prosessointi-, tallenn- sekä kommunikointipalvelut, joita tarvitaan järjestelmässä ajonaikaisesti. Taso sisältää kuvauksia myös järjestelmän fyysisistä elementeistä. (3.)

Näillä kolmella tasolla esitetyt elementit noudattavat yleistä geneeristä rakennetta, mutta niiden luonne ja rakeisuus vaihtelevat eri tasoilla.

Edellä luetellut ArchiMaten ydinkielen eri tasot kuvataan kolmesta eri näkökulmasta (aspect). Näkökulmadimensiot ovat seuraavat:

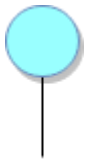
- Aktiivinen malli (Active Structure) esittää toimintojen tekijöitä kuten liiketoiminnallisia toimijoita tai järjestelmäkomponentteja.
- Käyttäytymismalli (Behavior) esittää toiminnallisuuksia, prosesseja, toimintoja, tapahtumia ja palveluita, joita suorittavat järjestelmän toimijat ja järjestelmäkomponentit.
- Passiivinen malli (Passive Structure) esittää kohteita, joihin toiminnallisuudet kohdistuvat. Näitä voivat olla liiketoimintatason kohteet tai tietomallit ja tietokohteet järjestelmätasolla. (3.)



KUVA 5. ArchiMate-viitekehys (3)

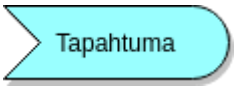
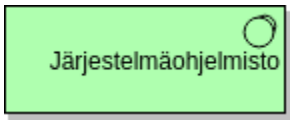
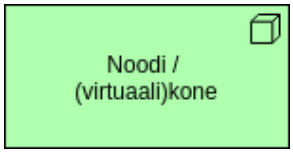
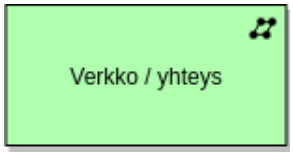
Luvussa 6 esitetyt järjestelmän arkkitehtuurikuvat noudattavat taulukon 1 mukaista notaatioita.

TAULUKKO 1. ArchiMate-notaatio

Elementti	**Kuvaus**	**Notaatio**
Toimija (Actor) Liiketoimintataso	Yleensä fyysinen henkilö tai henkilöryhmä, joka käyttää järjestelmää ja omistaa järjestelmän käyttötapauksen tai useita	
Komponentti (Component) Applikaatiotaso	Abstrakti osa järjestelmää. Sen sisältö ja koko riippuvat kontekstista, jossa se esitetään. Komponentti voi olla kuvattavan järjestelmä sisäinen tai se voi olla ulkoinen, löyhästi integroitu, helposti vaihdettavissa oleva ns. resurssi (35, linkki IV: Backing services)	
Rajapinta (Interface) Applikaatiotaso	Rajapinta kahden komponentin, osajärjestelmän tai järjestelmän ja toimijan välillä. Voi olla tekninen rajapinta tai toimijan käyttämä käyttöliittymä. Notaatiossa rajapinta voidaan esittää myös ”tikkari” ikonilla.	 
Palvelu (Service) Applikaatiotaso	Rajapinnan toteuttava tekninen palvelu	

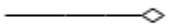
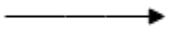
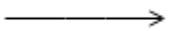
(taulukko jatkuu)

(taulukko jatkuu)

Tapahtuma (Event) Applikaatiotaso	Järjestelmän komponentin tai toimijan toiminnollaan aiheuttama tapahtuma järjestelmään	
Järjestelmäohjelmisto (System Software) Teknolokiataso	Järjestelmän tai sen ajoympäristön ohjelmisto; käyttöjärjestelmä tai muu ohjelmisto, jonka päällä järjestelmää ajetaan	
Noodi (Node) Teknolokiataso	Järjestelmän ja sen ajoympäristön yksi kone, esim. palvelin, jossa osa järjestelmän komponenteista ajetaan	
Toiminto (Function) Teknolokiataso	Järjestelmän toiminnallisuus	
Verkko (Networking) Teknolokiataso	Järjestelmän noodien ja hajautettujen komponenttien välinen yhteys, jonka avulla tekninen keskustelu tapahtuu	
Vaadittu rajapinta (Required Interface)	Järjestelmän toiminnon tarvitsema toisen komponentin tai ulkoisen järjestelmän toteuttama rajapinta	

(taulukko jatkuu)

(taulukko jatkuu)

Yksisuuntainen näkyvyys (Access one way)	Kahden komponentin tai toiminnon välinen yksisuuntainen yhteys tai kutsu	
Kahdensuuntainen näkyvyys (Access both way)	Kahden komponentin tai toiminnon välinen kaksisuuntainen yhteys tai kutsu; molemmat voivat kutsua toisiaan	
Realisaatio (Realization)	Komponentti toteuttaa (realisoi) jonkin toiminnon tai palvelun	
Assosiaatio (Association)	Yhdistetyt järjestelmän osat liittyvät toisiinsa. Liitynnän luonne on löyhä tai se jätetään toivotun abstraktiotason takia erittelemättä tarkemmin.	
Aggregaatio (Aggregation)	Koostuminen. Järjestelmän komponentti koostuu toisista komponenteista tai toiminnosta	
Käynnistys (Triggering)	Toiminto tai tapahtuma voi käynnistää toiminnon toisessa komponentissa	
Palvella (Serving)	Komponentti tai toiminto palvelee toista komponenttia tai toimintoa tai tuottaa haluttua toiminnallisuutta toiselle	

3 KÄYTETYT TEKNOLOGIAT

Tässä luvussa esitellään opinnäytetyön tekemisen aikana testattuja ja sen arkkitehtuuriin käyttöön otettuja teknologioita ja työkaluja. Koska teknologiaesittelyjä on internetin kautta tarjolla paljon, tässä luvussa keskitytään esittelemään vain kunkin teknologian ne oleelliset piirteet, jotka liittyvät tässä työssä toteuttavan järjestelmän vaatimuksiin ja niiden ratkaisemiseen. Lisäksi luvussa esitellään omia näkemyksiä ja kokemuksia teknologioista ja työkaluista.

3.1 Konttiteknologia

Vaikka kontit ja niihin liittyvät teknologiat ovat nousseet vasta viime vuosina esiin modernina ketteränä DevOps-mallia tukevana teknologiana (24), konttien ja ympäristöjen virtualisoinnin historia tähän päivään on pitkä (25). Kuten jo aikaisemmassa luvussa 2.1 todettiin, jo 60-luvulla keskuskoneita käytettiin erillisistä ”tyhmistä” terminaalikoneesta. Tämä tarkoittaa, että keskuskoneen vikaantuessa, mikään terminaali ei pystynyt jatkamaan toimintaansa. Tarvittiin siis keinoja sekä käyttäjätason (käyttöliittymän) että myös systeemitason prosessien erottamiseen. Luvussa 2.1 esitellystä client-server-arkkitehtuurista poiketen tässä ei tarkoiteta client- ja server-tasojen eriyttämisestä tietoverkon yli, vaan yhden tietokoneen ja sen käyttöjärjestelmätason prosessien erottamista.

Merkittävä kehitys virtualisoinnin suuntaan oli CHROOT (change root) -komennon kehittäminen. CHROOT-komento mahdollisti yhden prosessin ajoympäristön (käytännössä levyhakemiston) eriyttämisen itsenäiseksi muiden prosessien ajoympäristöstä. Unix-käyttöjärjestelmään komennon esittely vuonna 1982 Bill Joy. Kyseessä oli tuolloin Unixin ”seitsemäs painos”. (25.)

Toinen merkittävä hyppäys virtualisoinnin mahdollistajana oli Jail-komennon keksiminen 1990-luvulla ja lopulta sen esitteleminen vuonna 2000 FreeBSD-käyttöjärjestelmässä. Jail-komento toimii kuten edellä mainittu CHROOT, mutta juurihakemiston lisäksi se mahdollisti myös käyttäjätason sekä tietoverkon eriyttämisen itsenäiselle prosessille. (25.)

Muita merkittäviä keksintöjä konttien suuntaan olivat myös 2004 julkaistut Solaris-kontit (Solaris Zones), jotka tarjosivat edeltäjiänsä paremman mahdollisuuden eriyttää prosessille sen tarvitseman oman virtuaalitilan. 2006 Google julkaisi oman Control Groups (cgroups) -nimisen tekniikan prosessin virtuaalitilan hallintaan. Vuonna 2008 Control Group -tekniikka yhdistettiin Linuxin ytimeen Linux Containers (LxC) -projektissa, jolloin mahdollistettiin käyttöjärjestelmätason virtualisointi ja siten usean samanaikaisen Linux-ympäristön (kontin) ajamisen saman Linux-ytimen päällä. Jokaisella Linux-kontilla on siis oma prosessi ja verkkotilansa. (25.)

Nykyisin jo de facto -standardina tunnettu konttiteknologia Docker toteutettiin alun perin open-source -projektina LxC-tekniikalla vuonna 2013, mutta muutettiin hetkeä myöhemmin vuonna 2014 Googlen jo edellisenä vuonna tuomalle libcontainer-alustalle (25).

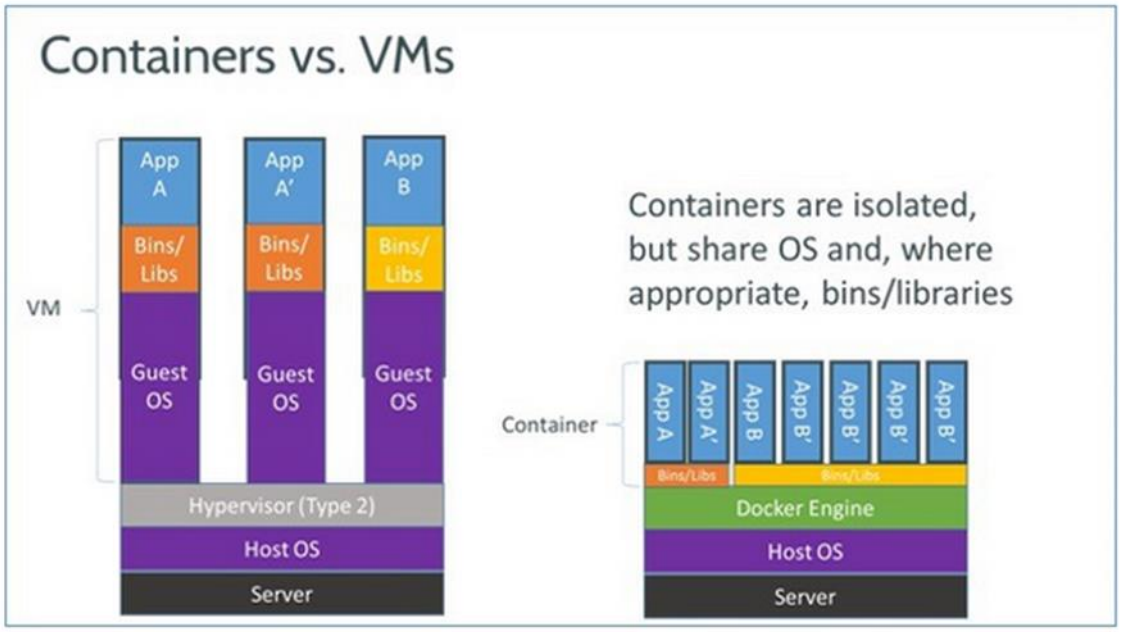
Tänä päivänä konttiteknologian käyttö on edelleen yleistymässä ja käyttö kasvamassa virtuaalikoneympäristöjen korvaajana. Arvioiden mukaan vuoteen 2020 konttiteknologiamarkkinat kasvavat nykyisestä 1,5 miljoonasta dollarista 3 miljardiin. Etenkin siirtyminen pilvipalveluihin ja -ympäristöihin kasvattaa konttiteknologioiden suosiota. Konttiteknologian merkittävimpänä haasteena yritykset näkevät tiedon tallentamisen konttiympäristössä. (26.)

3.2 Docker ja docker-compose

Kontti on siis tietokoneen käyttöjärjestelmän ytimen päällä toimiva virtualisoitu liikuteltava kokonaisuus. Konttiteknologiaa esiteltiin jo aikaisemmin luvussa 3.1.

Docker Inc. on amerikkalainen, vuonna 2010 perustettu yritys, joka on kehittänyt Docker-nimisen konttiteknologian (https://www.docker.com/). Docker on nykyisin lähes synonyymi konteille sen saaneen suuren suosion vuoksi (27).

Docker-kontti tehostaa palvelinresurssien käyttöä, koska esim. perinteisiin virtuaalikoneisiin verrattuna se käyttää tehokkaammin koneen resursseja sekä sisältää vähemmän ja kevyemmän kerroksellisuuden kuin täysin oman käyttöjärjestelmän sisältävä virtuaaliympäristö. Virtuaalikoneen ja konttiteknologian kerroksien eroja on kuvattu kuvassa 6.



KUVA 6. Virtuaalikoneen ja konttiteknologian kerroksien vertailu (27)

Docker on ympäristökuvista (image) luotujen konttien ajoympäristö. Ympäristökuvat voivat olla valmiita muiden tekemiä peruskuvia, joita voidaan käyttää pohjana omien sovelluskuvien rakentamiseen. Valmiita perus- ja sovelluskuvia on tarjolla paljon julkisissa repositoreissa, joista tunnetuin on Docker Inc. -yrityksen ylläpitämä Docker Hub -repositori ja sivusto (https://hub.docker.com/).

Esimerkkinä valmiista sovelluskuvasta on Docker yhteisön ja MySQL:n yhteistyössä ylläpitämä MySQL-kuva (https://hub.docker.com/_/mysql). Kyseessä on MySQL Community Edition -version. Yksinkertaisimmillaan MySQL-kuvan voi käynnistää kontiksi suoraan seuraavalla komennolla. Esimerkki olettaa, että koneelle on asennettu Docker.

$ docker run --name MyMysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:8

Esimerkkikomento lataa Docker Hubista saatavilla olevan mysql-nimisen sovelluskuvan (versio 8) ja käynnistää siitä koneelle MyMysql-nimisen kontin, jonka sisällä ajetaan MySQL-tietokantajärjestelmää. Käynnistyksen jälkeen MySQL-tietokantajärjestelmä on käytettävissä suoraan sivulla olevan ohjeistuksen mukaisesti.

Imagen sisältö määritellään Dockerissa ns. dockertiedoston (dockerfile) avulla. Kyseisen MySQL-sovelluskuvan Dockertiedosto on kuvattu liitteessä 1. Tiedoston sisällön riviltä 1 voidaan nähdä, että peruskuvaksi MySQL-sovelluskuvalle on valittu Debian-käyttöjärjestelmä. Joku on siis tehnyt Docker Hub -repositoryyn ”debian”-nimisen ympäristökuvan, joka rivillä 1 ladataan MySQL-sovelluskuvan pohjaksi ja kontin käyttöjärjestelmäksi. Riveillä 56-66 asennetaan dockertiedostossa varsinainen MySQL halutulla versiolla. Rivillä 76 tapahtuu MySQL-tietokantajärjestelmän prosessin käynnistäminen kontin sisälle.

Docker Compose on työkalu usean kontin muodostaman sovelluskokonaisuuden määrittelemiseen ja suorittamiseen. Docker konttien muodostama palvelukokonaisuus kuvataan ”docker-compose.yaml” -YAML tiedostossa. Esimerkki docker-compose tiedosta on kuvassa 7. Siinä muodostetaan kahden palvelun kokonaisuus. Ensimmäinen palvelu nimeltään ”web” rakennetaan hakemistossa olevan dockertiedoston perusteella (rivi 4). Web-palvelukuva voi olla esim. jokin HTTP-palvelin. Sitä emme tästä esimerkkitiedostosta näe.

Toinen palvelukuva, nimeltään ”redis”, muodostetaan suoraan julkisesta repositorystä saatavasta Redis-tietokantakuvasta. ”web”-palvelukuvan määrityksen rivillä 10–11 muodostetaan linkki ”web”- ja ”redis”-konttien välille, jolloin kontit ”näkevät” toisensa ja ”web”-kuvasta muodostettu kontti voi suoraan muodostaa yhteyden Redis-tietokantaan käyttämällä ”redis”-nimeä ”localhost”-nimen sijaan (hostname). Tämä määritelty kokonaisuus voi siis olla esim. pieni websivusto, joka käyttää tiedon tallentamiseen Redis-tietokantaa.

```
1: version: '3'
2: services:
3:  web:
4:   build: .
5:   ports:
6:   - "5000:5000"
7:   volumes:
8:   - .:/code
9:   - logvolume01:/var/log
10:   links:
11:   - redis
12: redis:
13:   image: redis
14: volumes:
15:  logvolume01: {}
```

KUVA 7. Esimerkki docker-compose.yaml -tiedostosta

3.3 Kubernetes

Kubernetes (kubernetes.io) on Googlen vuonna 2014 avoimeksi lähdekoodiksi julkaisema järjestelmä konttien ja niistä koostuvien ympäristöjen hallintaan. Kubernetesin avulla voidaan kontteja (kuten Docker kontteja) ”orkestroida” sille kuvattuun ympäristöön (kluster) määriteltyihin palvelimiin (noodeihin). Kubernetes luo klusterin sisällä noodien välille virtuaalisen verkon, jonka kautta noodien ja niissä ajettavat konttien liikenne ohjataan. Kubernetesin avulla voidaan esim. palvelun kuorman kasvaessa asentaa ja käynnistää lisää halutun kontin sovelluskuvan mukaisia palvelukontteja noodeihin ja muodostaa välityspalveluja (proxy), jotka jakavat kuormaa automaattisesti kaikille saman sovelluskuvan konteille. Tämä skaalautuminen voi myös tapahtua automaattisesti, kun Kubernetes havaitsee kuorman kasvavan sille asetetun raja-arvon yli.

Kubernetesin hallinnoimaa konttiympäristöä voidaan ohjata omalla cubectl Command Line Interface (CLI) -työkalulla ja käytettävät kontit sekä koko ajoympäristö kuvataan YAML-tiedostoina. Kuvassa 8 on yksinkertainen esimerkki yaml-tiedostosta, jossa muodostetaan ympäristöön yksi Kubernetes Pod -kokonaisuus. Pod on komponentti, joka sisältää yhden tai useamman kontin, niiden välisen loogisen verkon ja yhteisen tallennustilan. Perinteisiin ympäristöihin verrattuna pod on yksi looginen kone, jossa kontit näkevät toisensa yhteisellä host-nimellä *localhost*. Yksi palvelinnoodi voi sisältää useita pod-kokonaisuuksia. Kuvassa olevassa esimerkissä my-web-site-niminen pod sisältää kaksi konttia: frontend-nimisen nginx

HTTP-serverin (https://www.nginx.com/) sovelluskuvasta muodostettavan edustapalvelimen sekä backend-nimisen tomcat webpalvelimen (http://tomcat.apache.org/) sovelluskuvasta muodostettavan palvelimen.

Kyseinen Pod voidaan käynnistää Kubernetes-ympäristöön komennolla

```
> kubectl create -f pod.yaml
```

Ympäristön ja podia voidaan tarkastella seuraavalla komennolla, jolloin Kubernetes näyttää ympäristössä ajettavan podin ja siinä ajettavien konttien (tässä kaksi) tilan.

```
> kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
my-web-site   2/2       Running   0          14s
```

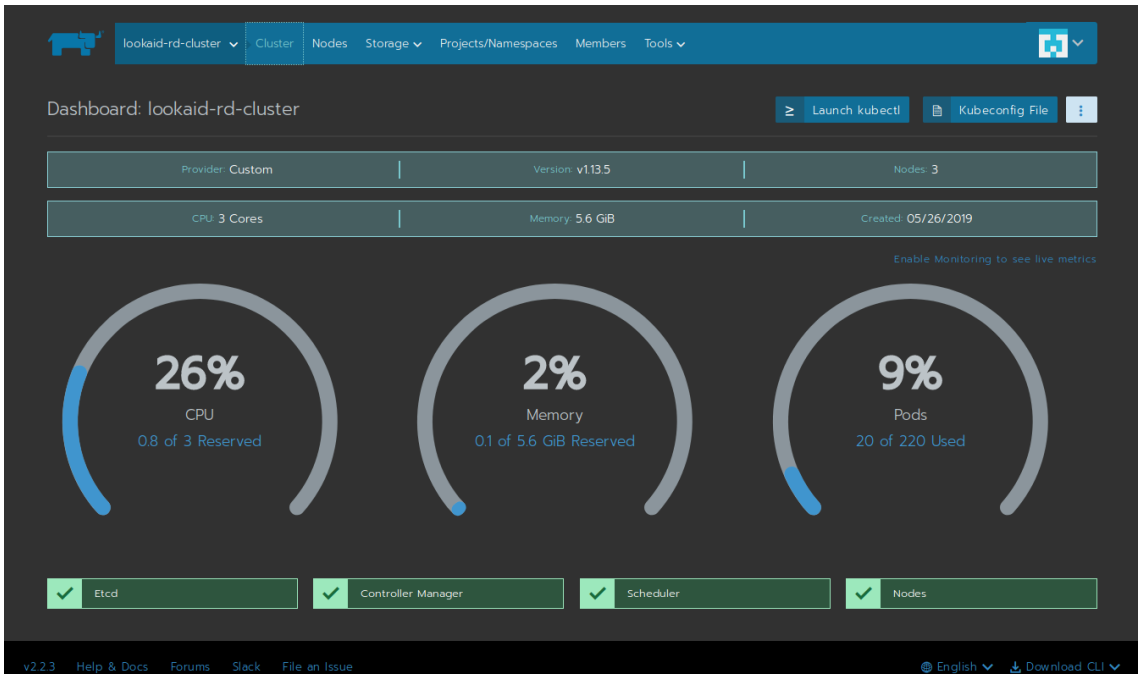
```
1: apiVersion: v1
2: kind: Pod
3: metadata:
4:  name: my-web-site
5:  labels:
6:    app: web
7: spec:
8:  containers:
9:    - name: front-end
10:     image: nginx
11:     ports:
12:       - containerPort: 80
13:   - name: back-end
14:     image: tomcat
15:     ports:
16:       - containerPort: 88
```

KUVA 8. Kubernetes konfiguraatiotiedosto (pod.yaml)

3.4 Rancher

Rancher R2 (https://rancher.com/products/rancher/) on järjestelmä ja alusta Kubernetes-klustereiden hallintaan. Se toteuttaa helppokäyttöisen graafisen käyttöliittymän Kubernetes klustereiden palvelinten ja konfiguraation hallintaan. Rancherin avulla voi hallinta useaa Kubernetes-klusteria samassa järjestelmässä, jolloin kehitys-, testi- ja tuotantoympäristöjä voidaan hallinta yhden käyttöliittymän kautta ilman manuaalista konfiguraatiotiedostojen (YAML) ylläpitoa ja cubectl (ks. luku 3.3) komentorivityökalua.

Kuvassa 9 on Rancher R2 -käyttöliittymä, jossa esitetään yhden klusterin dashboard-näkymä. Klusterissa on 3 noodia, joissa ajetaan yhteen kahtakymmentä Kubernetes podia. Dashboard sisältää myös kluster-ympäristön noodien kokonaistilaa kuvaavat mittarit ja Kubernetes-klusterin sisäisten komponenttien statukset.



KUVA 9. Rancher -käyttöliittymän cluster-näkymä

3.5 Elasticsearch

Elasticsearch (https://www.elastic.co/products/elastic-stack) (ES) on yksityishenkilön Shay Banonini kehittämä ja avoimen lähdekoodin hakukoneeksi 2000-luvun alussa julkaisema hajautettu hakutietokanta. Järjestelmää kehittää nykyisin vuonna 2012 perustettu Elasticsearch Inc. yritys ja sen tarjontaan kuuluvat nykyisin Elasticsearchin lisäksi myös lokitiedostojen hallinta- ja analysointityökalut Logstash (https://www.elastic.co/products/logstash) ja Kibana (https://www.elastic.co/products/kibana). Julkisuudessa näkyvin käyttötapaus ES-tuotteelle on ns. ELK-stack, joka tarkoittaa Elasticsearch-Logstash-Kibana-tuotteista rakennettua, lokien keräämiseen, indeksointiin ja hakuun sekä monitorointiin tarkoitettua järjestelmäkokonaisuutta (https://www.elastic.co/what-is/elk-stack).

ES:n avulla voidaan indeksoida tietoa tekstihakuja (full-text-index) sekä muun tyyppisiä hakuja varten. ES on käytännössä NoSql- ja skeematon tietokanta, joka toteuttaa järjestelmän integrointia helpottavan REST-rajapinnan. Suoran HTTP REST -rajapinnan lisäksi ES:ään on toteutettu helppokäyttöisiä client-kirjastoja mm. Java- ja Python -kielille. Tieto varastoidaan ES -hakukantaan json-dokumentteina.

ES:n hakumoottorin nopean indeksoinnin ja hajautetun arkkitehtuurin ansiosta sillä on hyvä virheensietokyky (fail-safety). ES:n indeksointiin voidaan konfiguroida tai toteuttaa räätälöidysti myös omia indeksointimalleja, joilla voidaan vaikuttaa indeksien tallennukseen. Esimerkkinä tällaisesta indeksianalysaattorista on ”Pattern Analyzer”, joka jakaa tekstin säännöllisten lausekkeiden avulla erillisiksi termeiksi indeksikantaan. Tämän opinnäytetyön kohteena olevassa järjestelmässä ei, ainakaan alkuvaiheessa, ollut tarvetta muille kuin oletuksena saataville indeksointi analysaattoreille.

ES hakumoottori sisältää myös tämän opinnäytetyön kohteena olevan järjestelmän vaatimuksiin seuraavanlaisia ominaisuuksia:

- nopea tiedon indeksointi
- hajautettu dokumenttien tallennus ja sitä kautta lähes reaaliaikainen tiedon haku
- erikoishaut, kuten ”näistä voisit olla myös kiinnostunut” (More-Like-This, MLT) haut sekä haettavien kohteiden sijaintitietoon (esim. koodinaatistoon) perustuvat haut (geospatial seach).

Tässä työssä saadut kokemukset ES:stä olivat positiivisia. Järjestelmä oli helppo integroida ja sen indeksointia ja sisältöä on helppo hallita REST-rajapinnan kautta. ES ei kuitenkaan toteuta täysin itsenäisesti ja suoraan hallinta- ja monitorointikäyttöliittymää kuten esimerkiksi seuraavassa luvussa esitelty Solr-tuote. Tämän lisäksi järjestelmän tietoturva-vaatimusten vaatimat autentikoinnit ja SSL-yhteydet eivät olleet tämän opinnäytetyön kohteena olevan järjestelmän teknologioiden valinnan aikana osa ES-ympäristön open-source-tuotetta vaan vaativat kaupallisen lisenssin ostamisen (https://www.elastic.co/blog/security-for-elas-

ticsearch-is-now-free). Koska kohteena olevan järjestelmän vaatimuksissa tietoturva on oleellinen, katsottiin tietoturvaominaisuuksien kaupallisuus tuossa vaiheessa esteeksi ES:n käyttöönotolle. Sen sijaan lokienhallintaa varten ELK-stack:n käyttöönotto on mahdollista myöhemmässä vaiheessa.

3.6 Solr

Koska tässä luvussa esiteltävä Sorl-järjestelmä (http://lucene.apache.org/solr/) on perusperiaatteiltaan samanlainen kuin edellisessä luvussa esitelty Elasticsearch (ES), kuvataan tässä luvussa lähinnä vain Solrin eroavaisuuksia ES-järjestelmään.

Solr on ES:n jälkeen yksi suosituimmista hakukonejärjestelmistä (https://db-engines.com/en/ranking). Kuten ES, Solr on myös nopea ja luotettava hakukone johtuen hajautetusta indeksointi- ja dokumenttikannasta, automaattisesta indeksien replikoinnista ja automaattisesta virheistä toipumisesta. Solr on myös täysin maksuton avoimen lähdekoodin järjestelmä, myös tietoturvaominaisuuksiltaan, toisin kuin ES oli järjestelmän kehitysvaiheen aikana.

Solr tuottaa kohteena olevan järjestelmään samat edut kuin ES (ks. luku 3.5), mielestäni jopa käytännössä havaittuna paremmin esim. More-Like-This -hakutulosten perusteella tarkasteltuna. ES:stä poiketen Solr sisältää oletuksena indeksien hallinta ja perus monitorointikäyttöliittymän. Sen sijaan Solrin konfigurointi oli ES:ää työläämpää ja vaikka Solriin on toteutettu java-client, sen integroiminen järjestelmään vaati jopa client-toteutuksen räätälöintiä integraation toiminnan mahdollistamiseksi.

Solrin täysi maksuttomuus, oletuksena tuleva hallintakäyttöliittymä ja hyvät hakuominaisuudet painoivat osaltaan hakuindeksijärjestelmän valintakriteereissä.

3.7 Apache Kafka

Apache Kafka (https://kafka.apache.org/) on LinkedInin kehittäjien toteuttama reaaliaikainen tapahtumapohjainen palvelinohjelmisto, jonka toiminallisuus perustuu perinteisen viestijonojärjestelmän (message queue) tapaan tilaaja/julkaisija (publish/subscibe) -malliin (22). Apache Kafkaa voidaan käyttää toteuttamaan reaktiivinen tapahtumapohjainen-arkkitehtuuri (luku 2.5) ja mikropalveluiden

transaktioiden hallintaan käytetyn Saga-mallin hallintakomponentti (luku 2.3). Tilaajat voivat tilata tapahtumia haluamastaan Kafkan tapahtumavirrasta, jota kutsutaan topiciksi.

Toisin kuin perinteisessä viestijonojärjestelmässä, jossa käsitelty jonon viesti poistetaan, Kafkassa jonon tapahtumat on tallennettu ja luettavissa aina halutusta historian kohdasta eteenpäin. Jokainen tapahtumavirran tilaaja (Kafka client) ylläpitää muistipaikkaa, jonka mukaisen tapahtuman se on viimeiseksi lukenut. Uusi tilaaja voi siten lukea tapahtuman kohdasta nolla alkaen niin haluttaessa tai aloittaa tapahtumien käsittelyn seuraavasta tulevasta tapahtumasta. Kafkalla toteutetussa tapahtumapohjaisessa arkkitehtuurissa (23) järjestelmän nykyhetken tila voidaan näin aina muodostaa uudelleen lukemalla järjestelmän tapahtumat alusta alkaen. Tämä malli parantaa järjestelmän saatavuusvaatimuksia, sillä hetkellisesti poissa käytöstä olevat tilaajat jatkavat tapahtumien käsittelyä siitä, mihin aikaisemmin jäivät.

Kafkan tapahtumavirta-mallin etuna perinteiseen jonotekniikkaan verrattuna ovat seuraavat ominaisuudet:

- Tapahtumavirta on ”ehtymätön”, kun taas perinteisten viestijonojen koko on kiinteä ja jono voi siten täyttyä ja lakata vastaanottamasta viestejä tai viesti saadaan vasta jonoon, kun siihen tulee tilaa.
- Tietovirrasta viestit voidaan lukea helposti uudelleen, kun taas jonosta ne poistetaan heti käsittelyn jälkeen.
- Edellisestä myös seuraa, että tapahtumia voidaan käsitellä usealla erilaista toiminnallisuutta sisältävillä itsenäisillä tilaajilla reaktiivisesti, kun perinteisessä viestijonossa viesti voidaan käsitellä vain kerran yhden tyyppisellä tilaajalla imperatiivisesti (katso luku 2.4 imperatiivisestä ja reaktiivisestä mallista).
- Tietovirran käsittelyn toiminnallisuutta on helppo laajentaa lisäämällä kuuntelijoita ja tapahtumakäsittelijöitä (processors) vaikuttamatta tietovirran tapahtumien luonnin tehokkuuteen.
- Kafka on jo lähtökohtaisesti suunniteltu horisontaalisesti skaalautuvaksi, kun taas viestijonototeutukset kuten RabbitMQ, skaalautuvat lähinnä vain vertikaalisesti. (28; 22.)

4 JÄRJESTELMÄN YLEISKUVAUS JA VAATIMUKSET

Tässä luvussa esitellään arkkitehtuurisuunnittelun kohteena oleva järjestelmä. Koska startup-yrityksen liiketoiminta- ja palveluidea ovat toistaiseksi suojattuja, toteutettava järjestelmä ja sen vaatimukset kuvataan tässä luvussa yleisin termein ja järjestelmän vaativat kyvykkyydet kuvaten.

Sekä järjestelmän yleiskuvaus ja vaatimukset esitellään seuraaviin teemoihin jaoteltuna:

- startup-yrityksen erityispiirteet
- monikanavaisuus
- monimuotoinen toiminnallisuus
- tietoturvallisuus ja tietosuoja
- suorituskyky, skaalautuvuus
- tekninen alusta (kuvataan vain vaatimusten osalta).

4.1 Yleiskuvaus

4.1.1 Startup-yrityksen erityispiirteet

Startup-yritys tarkoittaa nuorta, innovatiivista, tuotekehitykseen painottuvaa ja nopeaa kasvua tavoittelevaa yritystä (29). Yrityksen tuotekehitysvaiheessa tyypillisesti startup-yrityksen käytettävissä olevat raha-, aikataulu- sekä henkilöresurssit ovat rajalliset. Rajoitetuin resurssein tehtävän tuotteen kehitysvaiheen ja sen laadun ja resurssien hallinnan ongelmaa voidaan esittää kuvan 10 mukaisella projektihallinnan kolmioteorialla, jossa tuotteen laatudimensioon voidaan vaikuttaa kustannus-, aikataulu- ja toiminnallinen laajuus -rajoitteilla. Kolmioteorian mukaan muutos johonkin kolmesta esitetystä rajoitteesta tulee kompensoida muutoksena muihin rajoitteisiin tuotteen laadun heikentymisen välttämiseksi. (30.) Esimerkiksi startup-yrityksen tuotekehitysvaiheessa tuotteen toiminnallinen laajuus kasvaa helposti (ns. feature creep), jolloin aikataulua ja yleensä sitä kautta myös kustannuksia tulisi kasvattaa, jotta toteutettujen ominaisuuksien laatu ei kärsi. Näin ollen käytössä olevat rajalliset raha- ja aikaresurssit tulee kohdentaa ns. pienimpään mahdolliseen tuotteeseen (Minimal Viable Product, MVP)

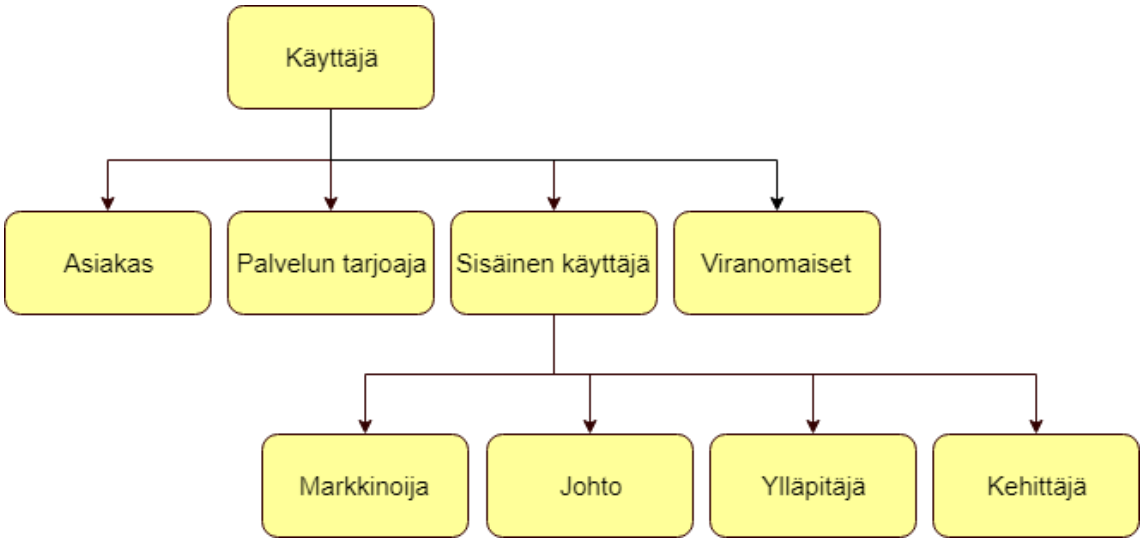
tehokkaasti. Minimaalisessa tuotteessa on toteutettuna vain ne tärkeimmät tuotteen ominaisuudet, joilla yritys pystyy tulemaan ulos markkinoille testatakseen kaikista epävarmimpia oletuksiaan tuotteeseen ja asiakkaiden käyttäytymiseen liittyen (31).

KUVA 10. Projektikolmio (mukaillen 30)

4.1.2 Monikanavaisuus

Tänä päivänä digitaalisia palveluita ja järjestelmiä käytetään kasvavassa määrin erilaisista laitteista ja kanavista (32). Perinteinen verkkosivuilla toteutettu verkkokauppa ei myöskään enää tavoita kaikkia potentiaalisia asiakkaita ja heidän ostokäyttäytymisensä on laajentunut muihinkin kanaviin (33).

Toteutettava järjestelmä on monikanavainen, millä tarkoitan, että järjestelmää ja sen toiminnallisuutta käytetään useasta eri frontend-alustasta kuten mobiililaitteista (puhelimet ja tabletit, natiivisovellukset ja selaimet), perinteisistä tietokoneista ja niiden selaimista sekä palvelualustan toiminnallisuutta hyödyntävistä eräajoista. Käyttökanaviin lasketaan myös eri sisällöntuottoalustat ja -mediat kuten sosiaalinen media (Facebook, Twitter) sekä raportointivälineet, kuten liiketoimintajohdon raportointityökalut. Järjestelmän toimijat, kuten palvelun tarjoajat sekä markkinoijat (kuva 11) hyödyntävät sosiaalisen median kanavia palvelunsa esiin tuomisessa. Myös Fonl Oy itse hyödyntää eri kanavia toteutettavan järjestelmän markkinoinnissa.



KUVA 11. Järjestelmän toimijat

4.1.3 Monipuolinen toiminnallisuus

Järjestelmä toimii palveluntarjoajien ja palveluiden hyödyntäjien kohtaamispaikkana ja tarjoaa rikkaan ja kokonaisvaltaisen toiminnallisuuden sekä käyttökokemuksen käyttäjilleen palveluiden ja niiden asiakkaiden hallintaan.

4.1.4 Tietoturvallisuus

Monipuolisen toiminnallisuuden saavuttamiseksi järjestelmään tulee kerätä perustietoja, myös tietosuojattua tietoa, käyttäjistä, etenkin palvelun tarjoajista. Kuvassa 11 kuvattiin järjestelmän tunnistetut käyttäjäroolit. Toiminnallisuuteen liittyy myös asiakkaiden ja palveluntarjoajien välisten ostosopimusten ja sopimusten mukaisten rahasummien tallentamista ja käsittelyä. Myös maksusuoritukset liikkuvat palvelun läpi, vaikka varsinaiset maksut tapahtuvat 3. osapuolten palveluita hyödyntäen.

4.1.5 Suorituskyky ja skaalautuvuus

Järjestelmän potentiaalinen käyttäjäkunta koostuu alkuvaiheessa kaikista Suomen yli 18-vuotiaista työikäisistä kansalaisista, myöhemmin ehkä jopa muidenkin maiden vastaavista käyttäjistä. Järjestelmälle on alustavien arvioiden mukaan alkuvaiheessa odotettavissa satoja yhtäaikaisia käyttäjiä.

4.2 Vaatimukset

4.2.1 Startup-yrityksen erityispiirteet

Yrityksen omistajat ovat sijoittaneet ja hankkineet rajallisen määrän alkupääomaa liikeideansa ja tuotteensa kehittämiseen. Koska startup-yrityksen alkuvaiheessa (ns. bootstrapping-vaihe) yrityksen haasteena ovat pieni pääoma ja vähäiset resurssit, on resurssien käyttö optimoitava tehokkaasti. Toimintansa alussa yritys keskittyy toteuttamaan liikeideaansa tukevan järjestelmän vain sen ehdottomasti vaadituin toiminnallisuuksin (ns. Mimimal Viable Product, MVP), jonka avulla yritys lähtee markkinoimaan ja testaamaan liikeideaansa ja järjestelmäänsä pienellä asiakas ja käyttäjämäärällä (1). Kun yrityksen ideasta löytyy alkuvaiheen jälkeen kasvumahdollisuutta, on tarve myös skaalata liiketoimintaa sekä järjestelmän kehitystyötä ylöspäin nopeasti ja helposti, ilman että edelleen kasvun alkuvaiheen vähäisiä resursseja kasvatetaan liikaa. Teknologinen skaalautuvuus digitaalisissa palveluissa tarkoittaakin kyvykkyyttä nostaa suorituskykyä ilman että järjestelmän alustaa rakennetaan joka kerta uudelleen (34, s. 149).

4.2.2 Monikanavaisuus

Järjestelmän perusta on natiivi mobiilisovellus sekä Android- että iOS-alustoille. Selainalustoihin ei ole määritelty rajoitusta, joten siihen lasketaan yleisimmät käytettävissä olevat selaimet kuten Firefox, Chrome ja Edge ja niiden yleisimmät suhteellisen tuoreet versiot. Web-käyttöliittymissä tulee huomioida myös mobiililaitteiden selaimet.

Monikanavaisuus huomioon ottaen järjestelmän tulee olla frontend-agnostinen, mikä tarkoittaa, että palvelualustatoiminnallisuus on oltava clientin toteutuksesta riippumatonta sekä standardi kaikille frontend-toteutuksille.

Monikanavaisuuden ei tule myöskään johtaa useampaan backend-toteutukseen. Yrityksen tulisi suunnitella ja toteuttaa pieniä tarkoituksenmukaisia sovelluksia eri kanaville ja hyödyntää noissa sovelluksissa yhteistä backend-toiminnallisuutta.

Poikkeukset tähän yleiskäyttöiseen malliin on toteutettava korkeintaan yksittäisinä backend-rajapintoina ja -toiminnallisuuksina.

4.2.3 Monipuolinen toiminallisuus

Käyttökokemukseltaan ja toiminnoiltaan monipuolisen järjestelmän kehittämisessä tulee hyödyntää joustavaa ja helposti laajennettavaa toiminnallisuutta sekä käyttökanava- että palvelualusta-arkkitehtuurissa. Tämä tarkoittaa nykyaikaisessa käyttäjän kaikkialla läsnäolevassa (omnipresent) ja digitaalisessa liiketoiminnassa monipuolisuutta sekä toteutustekniikoissa että integraatioratkaisuissa (32). Lisäksi alkuvaiheestaan startup-yrityksen toiminnan potentiaalinen laajentuminen sekä käyttäjien että ominaisuuksien suhteen vaatii mukautuvaa ja helposti laajennettavaa, sekä horisontaalisesti että vertikaalisesti skaalautuvaa arkkitehtuuria.

Taulukossa 2 on kuvattu järjestelmän ideointi- ja suunnitteluvaiheessa tunnistetut vaaditut kyvykkyydet. Myöhemmin tässä opinnäytetyössä esitellään nämä kyvykkyydet toteuttavat teknologiakomponentit.

TAULUKKO 2. Järjestelmän toiminnallisia vaatimuksia kyvykkyysalueittain

Kyvykkyysalue	**Toiminnallinen vaatimus**

Viestintä	Järjestelmä käyttäjien välinen kommunikaatio mahdollistettava seuraavasti:
	• keskustelut
	  ○ 1-to-1, 1-to-many
	  ○ 1-to-1 puhelut
	• 1-to-many notifikaatiot, sähköpostit ja tekstiviestit
	• 1-to-1 palautteet

Tapahtumavirta	Järjestelmän tulee toteuttaa ja pystyä esittämään tapahtumia seuraavasti: • käyttäjän toiminta ja tapahtumat järjestelmässä • järjestelmän tekniset tapahtumat

(taulukko jatkuu)

(taulukko jatkuu)

	• tapahtumiin reagointi ○ tapahtumista tulee pystyä käynnistämään lisää tapahtumia, kuten notifikaatioita (ks. Kyvyk-kyysalue ”viestintä”).
Raportointi	Järjestelmän tulee kerätä tietoa ja siitä tulee pystyä muodostamaan esimerkiksi seuraavan tyyppisiä raportteja ja statistiikkaa • johdon raportointi; esim. palveluiden kategoriat, niiden määrät ja käyttö, tehdyt sopimukset jne. • suosituimmat palvelut • suosituimmat hakusanat Datan laatu: • Kerätyn datan ja sen muodon tulee mahdollistaa myöhemmin myös muunlaiset raportit ja haut.
Haku	Palveluiden haku tulee mahdollistaa • tietokenttäkohtainen haku mm. palvelukategorioilla • avainsanahaku (full-text-search) • ”enemmän tällaista”-haut (More-Like-This -haut) • käyttäjän ja palvelun sijaintiin perustuvat haut • sumea (fuzzy) hakulogiikka käyttäjien kiinnostuksen kohdistamiseksi haluttuihin palveluihin (mainonta)

(taulukko jatkuu)

(taulukko jatkuu)

Profilointi	• järjestelmän tulee kerätä käyttäjistä tietoa, jonka perusteella voidaan mm. toteuttaa raportointia (ks. kyvykkyysalue ”Raportointi”) • profiloinnissa erotellaan kaksi käyttäjäryhmää: palvelun tarjoajat ja palvelun asiakkaat • järjestelmän tulee ottaa huomioon tietosuoja ja -turva kerätessään tietoa
Maksaminen	• maksun välitys • maksaminen • laskutus (sekä yrityksen oma laskutus että asiakkaiden laskuttaminen palveluiden käytöstä) • maksukuitit
Sisällönhallinta	• kuvapankki • tiedostojen hallinta (CV:t, luvat jne.) • kuvien ja tekstien automaattinen moderointi (tarkistaminen ja epäsopivien kuvien ja tekstien poistaminen)
Järjestelmän hallinta ja monitorointi	• mobiilisovelluksen etäkonfigurointi • mobiilisovelluksen käytön ja virheiden monitorointi ja analysointi • palvelinympäristön monitorointi • sekä mobiili- että palvelinympäristön lokien kerääminen ja analysointi • järjestelmän varmuuskopiointi ja palauttaminen
Markkinointi	• palvelumainokset • integraatiot sosiaaliseen mediaan

4.2.4 Tietoturvallisuus

Järjestelmän tietosisältö on pääosin avointa henkilödataa (ei yksilöivää), mutta niiden tietosuojan alaisten tietojen, joita järjestelmään tallennetaan, tulee noudat-

taa lain ja säädösten vaatimuksia. Myös asiakastietojen hallinnan toiminnallisuuksien tulee noudattaa säädöksiä kuten asiakkaan oikeus tulla ”unohdetuksi”, jolloin asiakastiedot ja asiakkaaseen liittyvä tieto poistetaan järjestelmästä kokonaan. Yksikönsuojan oleellisempana säädöksenä on EU-tason General Data Protection Requlation (GDPR) (https://eugdpr.org/).

Sopimukset ja raha-arvot eivät ole tietosuojan alaista tietoa, ellei niitä yhdistetä käyttäjiin ja heidän välisiinsä suhteisiin, mutta siitä huolimatta järjestelmässä käsiteltävä sellaiset datan sisällöt tulee suojata myös ulkopuolisten tahoilta käyttäen tietoturvallisia ja suojattuja yhteyksiä ja pääsynhallintaratkaisuja.

Potentiaalisesti isot käyttäjämäärät ja korkea käyttöastetavoite aiheuttavat vaatimuksen järjestelmän ja tiedon virheettömyydelle ja toipumiselle mahdollisista ongelmatilanteista.

Laajan käyttäjäkunnan itsenäinen sisällöntuotanto johtaa etenkin nykymaailman tilan huomioon ottaen myös palvelun tuottajan vastuuseen sisällön asianmukaisuudesta. Jatkuvan uuden sisällön virtaa on käytännössä mahdoton valvoa ilman automatiikkaa, ja tämä tulee huomioida järjestelmässä.

4.2.5 Suorituskyky ja skaalautuvuus

Toiminnan alun järjestelmäkehitysvaiheessa järjestelmälle ei aseteta vielä suuria suorituskyky- ja skaalautuvuusvaatimuksia käyttäjämäärien ollessa kohtuullisia tai jopa hyvin pieniä. Toiminnan ja käyttäjämäärän kasvaessa tulee järjestelmän suorituskykyä pystyä kasvattamaan nopeasti ja kustannustehokkaasti ilman, että järjestelmän alustaa rakennetaan joka kerta uudelleen (34, s. 149).

4.2.6 Tekninen alusta

Tässä luvussa luetellut teknisen alustan ja toteutuksen vaatimukset perustuvat alun perin palveluna tarjottavan (Platform as a service, PaaS) Heruku-nimisen alustan kehittäjien määrittelemään ja myöhemmin Adam Wigginsin dokumentoimaan menetelmään ”Twelve-Factor App”. Menetelmä on tarkoitettu lähinnä internetissä palveluna tarjottavien sovellusten (Software as a service, SaaS) toteutukseen. (35.)

Menetelmän mukaisista vaatimuksista on jätetty pois järjestelmän kehitystyössä käytettyihin koodaustyökaluihin, riippuvuuksien hallintaan ja lähdekoodinhallintavälineisiin liittyvät vaatimukset.

Menetelmän mukaiset järjestelmän teknisen alustan vaatimukset ovat seuraavat:

- järjestelmän kehitys-, testaus- ja tuotantoympäristöjen ympäristökohtaisteh asetusten on oltava konfiguroitavissa ilman kooditason muutoksia (35, linkki III: Config)
- teknisten riippuvuuksien ja integraatioiden järjestelmän sisäisten ja ulkoisten komponenttien ja palveluiden välillä on oltava löyhiä ja konfiguroitavissa (35, linkki IV: Backing services)
- toimintavarmuus, skaalautuvuus ja tehokas toipuminen virhetilanteista on mahdollistettava toteuttamalla itsenäisiä ja tilattomia toiminnallisuuksia ja prosesseja (35, linkki VI: Processes)
- hyvän toimintavarmuuden aikaan saamiseksi prosessit tulee pystyä käynnistämään ja uudelleenkäynnistämään nopeasti ja ajamaan alas hallitusti (35, linkki IX: Disposability)
- kehitys-, testaus- ja tuotantoympäristön tulee vastata toisiaan ja samaa koodia pitää pystyä ajamaan sellaisenaan kaikissa ympäristöissä (35, linkki X: dev/prod parity)
- järjestelmän tulee kerätä lokitietoa häiritsemättä itse järjestelmän toimintaa ja suorituskykyä (35, linkki XI: logs).

5 ARKKITEHTUURISET JA TEKNOLOGISET VALINNAT

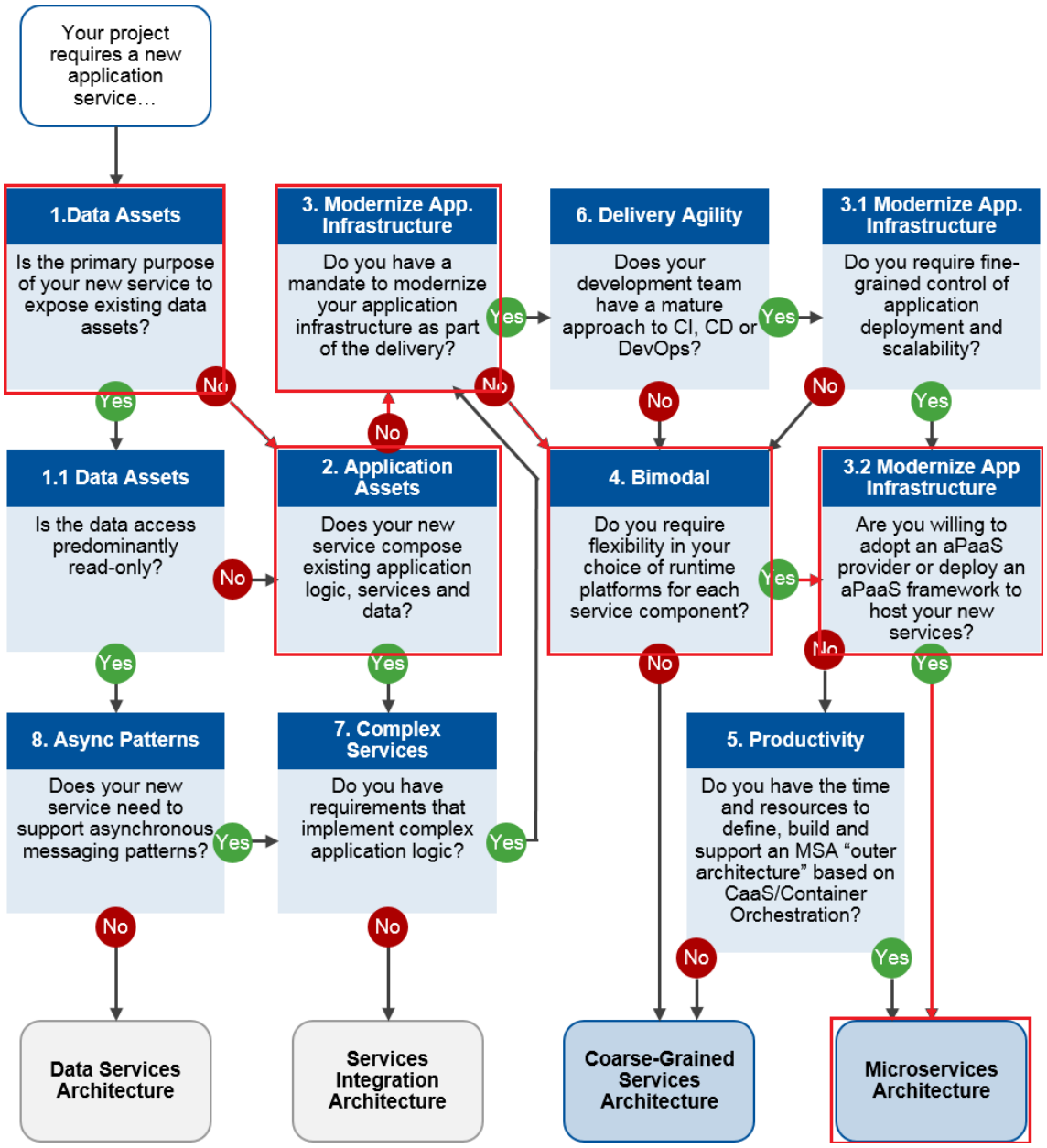
Toteuttavan järjestelmän arkkitehtuurin tulee toteuttaa tarvittavat kyvykkyydet järjestelmän toiminnallisuudelle asetettujen vaatimusten täyttämiseksi. Tämän opinnäytetyön kohteena olevan järjestelmän vaatimukset esiteltiin luvussa 4.2.

Nykymaailman digitalisaation kehityksen ja kohteena olevan järjestelmän vaatimukset huomioon ottaen ei ole perusteltua valita arkkitehtuuria ja teknologioita vanhaan monoliittiseen arkkitehtuurin perustuen. Monoliittisuuden sijaan valinnoissa on syytä tarkastella helposti laajennettavaa ja skaalautuvaa (myös tarvittaessa alaspäin skaalautuvaa) arkkitehtuuria. Tähän suuntaan teknologiavalinnoissa puoltavat vaatimukset monipuolisesta toiminnallisuudesta, käyttäjämäärän odotetusta laajentumisesta sekä kehitystyön tehokkaasta skaalausmahdollisuudesta.

5.1 Palveluarkkitehtuuri

Sopivaa palveluarkkitehtuuria suunniteltaessa voidaan hyödyntää kuvassa 12 esitettyä Gartnerin työkalua (36). Siinä päättelypolun avulla etsitään järjestelmän ei-toiminnallisten vaatimusten perusteella sopivaa palveluarkkitehtuuria. Työkalu ja päättelykriteerit eivät kuitenkaan ota huomioon palveluympäristöjen kustannuksia ja ympäristöjen rakentamiseen vaadittua aikaa, jotka ovat startup-yritykselle rajalliset. Tämä rajoite huomioiden työkalulla voidaan silti arvioida järjestelmän optimaalista tavoitearkkitehtuurimallia.

Kuvassa 12 ja taulukossa 3 on esitelty suunniteltavan arkkitehtuurin valintakriteerit ja päätöspisteet Gartnerin työkalua soveltaen.



KUVA 12. Työkalu palveluarkkitehtuurin valintaan. Arkkitehtuurivalinnan päättelypolku korostettu punaisella. (mukaillen 36)

TAULUKKO 3. Palveluarkkitehtuurimallin valintakriteerit päättelypolussa

Kriteeri	**Vastaus**	**Perustelu**
Onko palvelun päätarkoitus tuoda olemassa olevaa tietoa muiden käytettäväksi?	Ei	Järjestelmä on uusi eikä olemassa olevaa tietoa ole
Tuoko palvelu olemassa olevien järjestelmien toiminnallisuutta, palveluita ja dataa muiden käytettäväksi?	Ei	Järjestelmään ei integroida yrityksen olemassa olevia järjestelmiä
Tuleeko järjestelmäympäristöä uudistaa kehityksen aikana?	Ei	Ei vanhaa järjestelmää pohjana
Vaaditaanko palvelualustojen valinnassa joutavuutta?	Kyllä	Järjestelmän monikanavaisuus ja monipuolinen toiminnallisuus vaativat joustavuutta palveluiden toteutuksessa ja palvelualustoissa, jotta ne voidaan toteuttaa parhaiten sopivilla ja kustannustehokkaimmilla tekniikoilla
Onko mahdollista ottaa käyttöön application-as-a-service (aPaaS) -toimittajan tarjoama ympäristö tai asentaa oma aPaaS-ympäristö palveluiden alustaksi?	Kyllä	aPaaS-toimittajan käyttö on start-up-yrityksen alkuvaiheen vajavaisten työvoimaresurssien ja kustannusten syistä hyvin todennäköistä. Myöhemmin toiminnan ja resurssien kasvaessa voidaan myös tehdä päätös oman ympäristön hankinnasta

Työkalun päättelypolun lopputuloksen ja ei-toiminnallisten vaatimusten perusteella mikropalveluarkkitehtuuri olisi tässä tapauksessa järkevin vaihtoehto startup-yritykselle tehtävän järjestelmän arkkitehtuuriksi.

Laajan ja heterogeenisen käyttäjäjoukon tuottaman sisällön jatkuva analysointi, raportointi ja valvonta tulee automatisoida modernien algoritmien ja tekoälytoteutusten avulla. Palveluorientoitunut arkkitehtuurin valinta puoltaa myös tätä tarvetta (32, s. 13–14).

5.2 Toiminnalliset vaatimukset

Seuraavissa taulukoissa 4–13 on näkemys luvussa 4.2.3 esitetyissä toiminnallisissa vaatimuksissa valittuine teknologioineen ja perusteluineen. Perusteluissa mainituilla ”ilmainen”-termeillä tarkoitetaan että itse tuote on ilmainen. Huomioon ei siis oteta näissä tapauksissa kehitys-, ympäristö- ja ylläpitokustannuksia.

Taulukko 4. Teknologiavalinnat, vaatimusalue: viestintä

Viestintä	
Keskustelut	Applozic chat (https://www.applozic.com/) • valmiit SDK:t mobiilisovellusintegraatiolle (iOS, Android) • maltillinen alkuvaiheen hinnoittelu • käyttöliittymän kustomointimahdollisuus • paljon ominaisuuksia, ml. chatbot integraatiot Natiivit iOs:n ja Androidin puhelu- ja tekstiviesti-sovellukset • integroitavissa helposti suoraan sovellukseen • hyvä ja integroitu käyttäjäkokemus

(taulukko jatkuu)

(taulukko jatkuu)

Notifikaatiot	Firebase Cloud Messaging (https://firebase.google.com/docs/cloud-messaging) • maltillinen alkuvaiheen hinnoittelu • mukana jo teknologiapinossa Android-mobiilialustan kautta • yksinkertainen käyttökokemus • Java-client (Admin SDK)
Sähköpostit	Google Mail • alkuvaiheessa ilmainen • helppo integroida lähtevän sähköpostin palvelimeksi
Palautteet ja kyselyt	Typeform (https://www.typeform.com/) • hinnoiteltu maltillisesti • aikaisemmat hyvä kokemukset soveltuvuudesta tarkoitukseen

Taulukko 5. Teknologiavalinnat, vaatimusalue: tapahtumavirta

Tapahtumavirta	
Yksittäisen käyttäjän toiminta järjestelmässä Järjestelmän tekniset tapahtumat Tapahtumiin reagointi	Apache Kafka • ilmainen • tukee reaktiivista integraatiomallia • itsenäinen ja helposti skaalautuva • suorituskykyinen

Taulukko 6. Teknologiavalinnat, vaatimusalue: raportointi

Raportointi	
Raporttien tuotto Statistiikkojen laskenta Datan laatu	MySQL (https://www.mysql.com/) • ilmainen community edition jota voidaan hyvin käyttää ainakin järjestelmän kehitys ja testausvaiheessa • laadukas ja paljon käytetty relaatiokantajärjestelmä • relaatiokanta helpottaa ja tehostaa datan hallintaa ja käsittelyä • saatavilla valmiita integraatiokomponentteja mm. Java-alustoille Solr (luku 3.6) • ilmainen • sisältää valmiin indeksien hallinta- sekä monitorointikäyttöliittymän • tehokas • räätälöitävä (mm. indeksien käsittely prosessoreilla) • sisältää valmiita statistiikkahakufunktioita

Taulukko 7. Teknologiavalinnat, vaatimusalue: haku

Haku	
Tietokohtainen haku Avainsanahaku More-Like-This-haku Lokaaliin perustuva haku (spatial search)	Solr (luku 3.6) Kuten edellisessä taulukossa ”Raportointi”, lisäksi: • sisältää valmiita hakufunktioita, kuten M-L-T-haku ja spatial search • mahdollistaa monipuoliset hakufunktiot, sumeat hakulogiikat kuten myös kenttäkohtaiset haut

Taulukko 8. Teknologiavalinnat, vaatimusalue: profilointi

Profilointi	
Käyttäjätiedot Tietoturvallisuus	Ei varsinaista omaa teknologiaa. Käytetään MySQL tietokantaa tiedon hallintaan. Tietoja käsitellään vain salatun yhteyden kautta (SSL, HTTPS)

Taulukko 9. Teknologiavalinnat, vaatimusalue: maksaminen

Maksaminen	
Maksaminen ja maksun välitys Laskutus Kuitit	Stribe (https://stripe.com/) • maltillinen hinnoittelu • mahdollisuus integroida mm. chattiin • Rest API -rajapinta

Taulukko 10. Teknologiavalinnat, vaatimusalue: sisällönhallinta

Sisällönhallinta	
Kuvat ja tiedostot	Amazon S3 (https://aws.amazon.com/s3/) • maltillinen hinnoittelu • alkuvaiheen valinta olemassa olevan osaamisen takia Vaihtoehto: Firebase Cloud Storage (https://firebase.google.com/products/storage)
Moderointi Kuvat	Google Vision API (https://cloud.google.com/vision/) • maltillinen alkuvaiheen hinnoittelu • Googlen tuotteet jo vahvasti mukana teknologiavalinnoissa Vaihtoehto: Amazon Rekognition (https://docs.aws.amazon.com/rekognition/latest/dg/images-s3.html) • integroituu S3 Bucket -järjestelmään

(taulukko jatkuu)

(taulukko jatkuu)

Tekstit	Solr (luku 3.6): järjestelmän omat tekstimuotoinen tieto kuten palvelukuvaukset ja nimet tallennetaan jo valmiiksi Solr-hakuindeksiin.

Taulukko 11. Teknologiavalinnat, vaatimusalue: järjestelmän hallinta ja monitorointi

Järjestelmän hallinta ja monitorointi	
Mobiilisovelluksen etäkonfigurointi	Remote Config (https://firebase.google.com/docs/remote-config) • mukana jo teknologiapinossa Android-mobiilialustan kautta
Mobiilisovelluksen monitorointi	Firebase Crashlytics • teknologiapinossa jo mukana kehitys- ja testausvaiheen virheiden raportointiin ja hälytyksiin
Mobiilisovelluksen lokien keräys ja analysointi	Firebase Crashlytics: kuten yllä
Palvelinympäristön monitorointi	Rancher (luku 3.4) • ilmainen • helppokäyttöinen • koko klusterin / usean klusterin hallinta samasta paikkaa

(taulukko jatkuu)

(taulukko jatkuu)

	• tuottaa perusmonitorointivälineet • hälytykset integroitavissa mm. pikaviesti-miin chatbotin avulla
Palvelinympäristön lokien ke-räys ja analysointi	Elasticsearch + Kibana (luku 3.5) • ilmainen • helppo integroida Kubernets-clusteriin • suora Rancher-tuki
Järjestelmän varmuuskopi-ointi ja palauttaminen	Palvelinympäristötoimittajan palvelu MySQL-kannan peilaus toiseen slave-kantaan

Taulukko 12. Teknologiavalinnat, vaatimusalue: markkinointi

Markkinointi	
Mainostaminen	Ei varsinaista omaa teknologiaa. Käytetään hy-väksi sähköistä mediaa ja etenkin sosiaalista mediaa.

Taulukko 13. Teknologiavalinnat, vaatimusalue: muut tekniset vaatimukset

Muita teknisiä vaatimuksia	
Dynaamiset linkit	Firebase Dynamic Links • teknologiapinossa jo mukana

(taulukko jatkuu)

(taulukko jatkuu)

Docker-imagen jakelu	Google Could Registry (https://cloud.google.com/container-registry/) • maltillinen hinnoittelu • Googlen tuotteet jo vahvasti mukana tekno-logiavalinnoissa

6 ARKKITEHTUURI

Tässä luvussa kuvataan järjestelmän arkkitehtuuri ja sen komponentit. Kuvaamisessa käytetään The Open Groupin ArchiMate -mallinnusstandardin (3) notaatioita. Komponenttien kategorisoinnissa noudatetaan ArchiMate -määrityksiä.

Arkkitehtuurin kuvaus on tehty noudattaen Application Component Model (0-n) mallinnustapaa, joka koostuu eri kuvaustasoista seuraavasti:

- taso 0: kohdejärjestelmä kuvataan mustana laatikkona (blackbox) ja esitetään järjestelmän integraatiot sen ympäristöön
- taso 1: kohdejärjestelmä avataan ja kuvataan sen sisäiset päämoduulit ja niiden väliset integraatiot (whitebox)
- taso 2: kohdejärjestelmän päämoduulit avataan ja niiden sisäiset komponentit kuvataan (4).

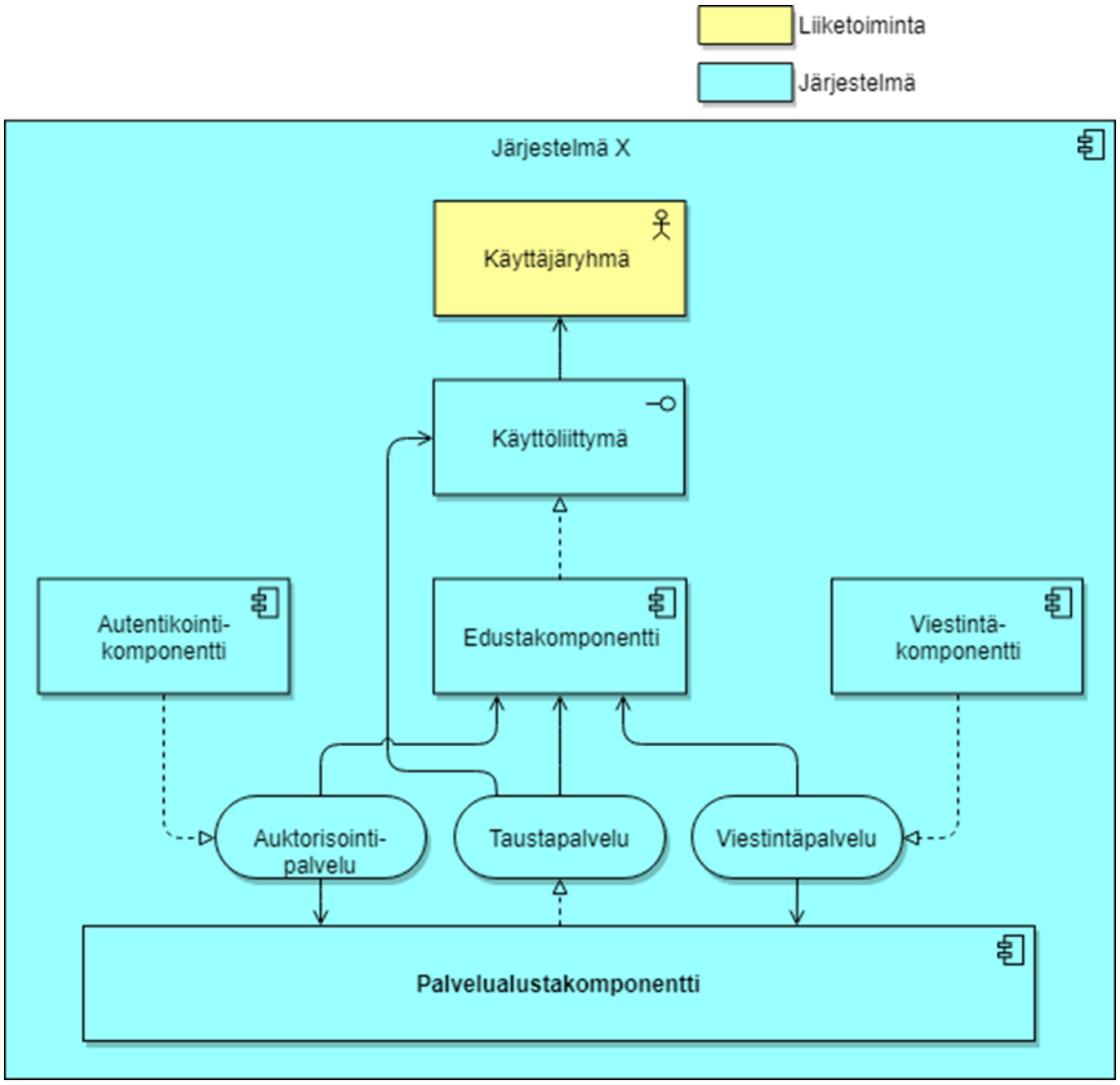
Arkkitehtuurin kuvaus rajataan järjestelmän palvelualustaosuuteen jättäen pois käyttökanavat ja niiden rakenteet. Kanavat kuvataan vain osana palvelualustan ympäristöä tasolla 0.

Kuvauksessa tarkennetaan opinnäytetyön pääkohteen eli palvelukomponentin sisäinen rakenne lisäksi toiminnallisena näkymänä (luku 6.4). Näiden lisäksi kuvataan vielä järjestelmän asennusnäkymä (luku 6.5), josta käy ilmi fyysiset komponentit ja esiteltyjen tasokomponenttien sijoittelu.

6.1 Komponenttimallin taso 0

Tässä luvussa kuvataan opinnäytetyön kohteena olevan järjestelmän ja komponenttimallin (4) taso 0.

Kuvassa 13 tämän arkkitehtuurikuvauksen kohteena oleva komponentti, palvelualusta (alin tummennettu komponentti), esitetään ”mustana” laatikkona ja kuvataan vain kohdekomponentin liittymät ulospäin. Kuva ei rajoitu kuvaamaan liittymiä vain yhteen ulkopuoliseen komponenttitasoon asti. Järjestelmän toiminnallisuuden kokonaiskuvan saamiseksi siinä kuvataan myös kohdekomponentin ulkopuolisten komponenttien liittymät niiden ulkoisiin komponentteihin.



KUVA 13. Järjestelmän arkkitehtuuri, taso 0

Taulukossa 14 on esitelty kukin kuvan 13 sisältämä komponentti ja sen toiminnallisuus järjestelmässä.

TAULUKKO 14. Komponenttitason 0 komponenttien kuvaus

Komponentti	**Kuvaus**
Käyttäjäryhmä	Järjestelmän käyttäjä, käyttäjäryhmä tai järjestelmän vaatimuksiin vaikuttava muu toimija tai toimijaryhmä. Järjestelmän toimijat on esitetty jo aikaisemmin luvun 4.1.2 kuvassa 11.
Käyttöliittymä	Järjestelmän käyttäjien käyttöliittymärajapinta. Käyttöliittymä voi olla järjestelmään toteutettu mobiilikäyttöliittymä, mutta myös esim. sosiaalisen median tai muun kolmannen osapuolen toteuttama käyttöliittymä jossa on integraatio palvelualustan taustapalveluihin.
Edustakomponentti	Järjestelmään toteutettu komponentti joka toteuttaa (realisoi) järjestelmän mobiili ja web käyttöliittymät
Palvelualustakomponentti	Varsinainen tämän opinnäytetyön kohteena oleva taustajärjestelmä joka toteuttaa tarvittavat palvelut edustakomponentin ja kolmansien osapuolten käyttöliittymiä varten
Taustapalvelu	Palvelualustakomponentin toteuttamat palvelut edustakomponenttia ja kolmansien osapuolten käyttöliittymiä varten
Autentikointikomponentti	Käyttäjän tunnistautumisen (autentikointi) ja valtuuttamisen (auktorisointi) toteuttavat komponentti. Kolmannen osapuolen tarjoama toiminnallisuus.

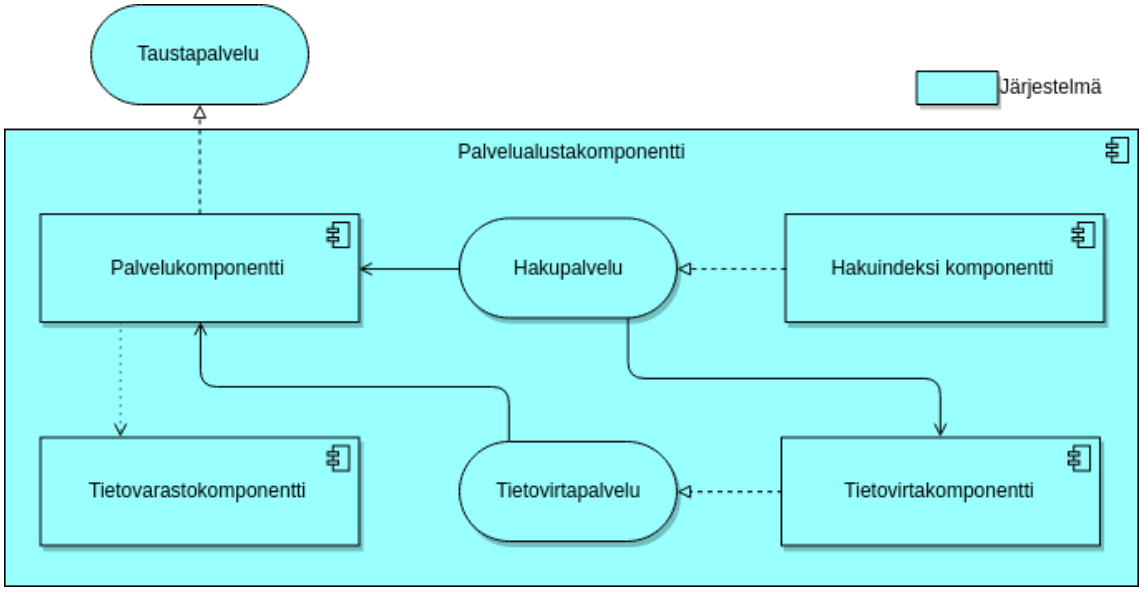
(taulukko jatkuu)

(taulukko jatkuu)

Auktorisointipalvelu	Autentikointikomponentin toteuttama palvelu, jonka kautta käyttäjän valtuutus voidaan toteuttaa palvelualustan toiminnallisuuteen
Viestintäkomponentti	Järjestelmä ja käyttäjien välisen viestinnän (esim. push-notifikaatiot tai sähköpostit) toteuttava komponentti
Viestintäpalvelu	Viestintäkomponentin toteuttama palvelu, jonka kautta palvelualusta voi lähettää viestejä järjestelmässä

6.2 Komponenttimallin taso 1

Tässä luvussa kuvataan kohteena olevan järjestelmän ja varsinaisen opinnäytetyön kohteena olevan palvelualustakomponentin rakenne ja toiminnot komponenttimallin (4) tasolla 1 (kuva 14).



KUVA 14. Järjestelmän arkkitehtuuri, taso 1

Taulukossa 15 on esitelty kukin kuvan 14 sisältämä komponentti ja sen toiminnallisuus järjestelmässä.

TAULUKKO 15. Komponenttitason 1 komponenttien kuvaus

Komponentti	**Kuvaus**
Palvelualustakomponentti	Kuvattu luvussa 6.1 , taulukossa 14
Palvelukomponentti	Palvelualustan taustapalveluiden toteuttava komponentti. Jokainen taustapalvelu on oma erillinen mikropalvelukomponenttinsa, kuten luvussa 6.5 kuvataan. Mikropalveluarkkitehtuuria esiteltiin luvussa 2.2.
Taustapalvelu	Kuvattu luvussa 6.1 , taulukossa 14
Tietovarastokomponentti	Tiedon tallennuksen toteuttava komponentti. Käytännössä tietokanta.
Hakuindeksikomponentti	Hakupalvelut toteuttava komponentti
Hakupalvelu	Palvelu jonka kautta palvelukomponentit voivat integroitua hakuindeksiin. Palvelukomponentti voi palvelun kautta sekä hakea että tallentaa tietoa hakuindeksiin. Hakupalvelun kautta myös tietovirta voi syöttää tietoa hakuindeksiin. Tällainen käyttötapaus on esimerkiksi tapahtumien auditlogin tallentaminen hakuindeksiin.
Tietovirtakomponentti	Tietovirtapalvelut toteuttava komponentti

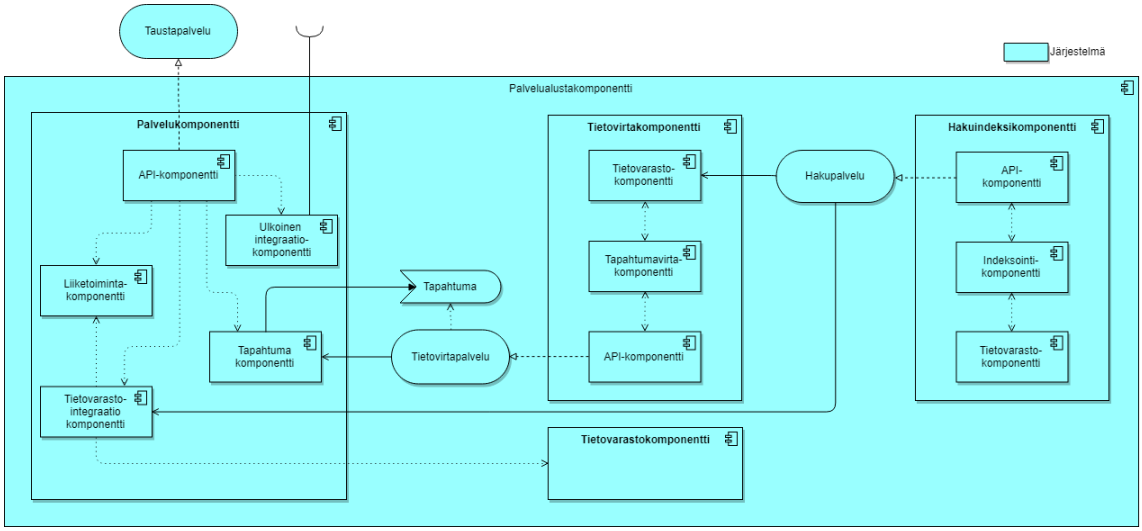
(taulukko jatkuu)

(taulukko jatkuu)

Tietovirtapalvelu	Palvelu, jonka kautta palvelukomponentit voivat integroitua tietovirtaan. Palvelukomponentit voivat sekä hakea tietoa että tallentaa tapahtumia tietovirtaan.

6.3 Komponenttimallin taso 2

Tässä luvussa avataan kunkin palvelualustakomponentin sisäisen komponentin rakenne komponenttimallin (4) tasolla 2 (kuva 15).



KUVA 15. Järjestelmän arkkitehtuuri, taso 2

Taulukossa 16 on esitelty kukin kuvan 15 sisältämä komponentti ja sen toiminnallisuus järjestelmässä.

TAULUKKO 16. Komponenttitason 2 komponenttien kuvaus

Komponentti	**Kuvaus**
Taustapalvelu	Kuvattu luvussa 6.1 , taulukossa 14
Palvelukomponentti	Kuvattu luvussa 6.2, taulukossa 15
• API-komponentti	Taustapalvelun Application Programming Integration (API) -rajapinnan toteuttava komponentti
• Ulkoinen integraatiokomponentti	Integraation ulkoisiin järjestelmiin toteuttava integraatio (esim. auktorisointipalveluun, ks. luku 6.1). Ulkoisen järjestelmän tulee toteuttaa rajapinta, johon voidaan integroitua.
• Liiketoimintakomponentti	Liiketoiminnan tietomallin ja toiminnallisuuden toteuttava komponentti. Tietomallia ja liiketoimintalogiikkaa käytetään API-rajapinnoissa tiedon deserialisointiin ja tiedon tarkistuksiin. Tietomallia käytetään tiedon tallentamiseen tietovarastoon.
• Tapahtumakomponentti	Tapahtumien tallennuksen ja tapahtumiin reakoivan toiminnallisuuden toteuttava komponentti (ks. luvut 2.4 ja 2.5)
• Tietovarastointegraatiokomponentti	Tietokanta-integraation toteuttava komponentti
Tietovirtapalvelu	Kuvattu luvussa 6.2, taulukossa 15

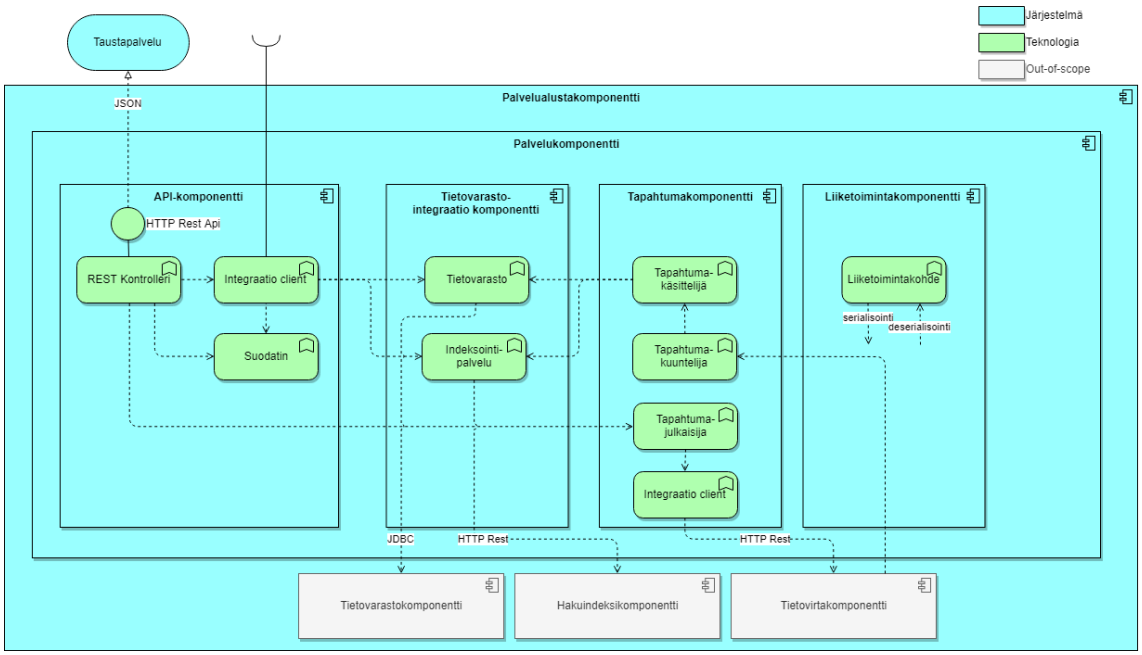
(taulukko jatkuu)

(taulukko jatkuu)

Tietovirtakomponentti	Kuvattu luvussa 6.2, taulukossa 15
• Tapahtumavirta-komponentti	Tapahtumavirta-toiminnallisuuden toteuttava komponentti
• Tietovarastokomponentti	Tapahtumavirtakomponentin tapahtumien tallennuksen toteuttava komponentti
• API-komponentti	Tietovirtapalvelut toteuttava komponentti
Tapahtuma	Tapahtumakomponentin emittoima, järjestelmän tai käyttäjän toiminnasta syntyvä tapahtuma
Tietovarastokomponentti	Kuvattu luvussa 6.2, taulukossa 15
Hakuindeksikomponentti	Kuvattu luvussa 6.2, taulukossa 15
• API-komponentti	Hakupalvelut toteuttava komponentti
• Indeksointikomponentti	Hakuindeksiin tallennettavan tiedon indeksoiva komponentti
• Tietovarastokomponentti	Hakuindeksin tiedon tallentava komponentti
Hakupalvelu	Kuvattu luvussa 6.2, taulukossa 15

6.4 Toiminnallinen näkymä

Tämän luvun kuvauksessa avataan vielä palvelukomponentin sisältämät toiminnallisuudet ja niissä käytetty tekniikka (kuva 16).



KUVA 16. Järjestelmän arkkitehtuuri, toiminnallinen näkymä

Taulukossa 17 on esitelty kukin kuvan 16 sisältämä komponentti ja sen toiminnallisuus järjestelmässä.

TAULUKKO 17. Palvelukomponentin toiminnallisen kuvauksen komponentit

Komponentti	**Kuvaus**
Palvelualustakomponentti	Kuvattu luvussa 6.1, taulukossa 14
Palvelukomponentti	Kuvattu luvussa 6.2, taulukossa 15
API-komponentti	Kuvattu luvussa 6.3, taulukossa 16
• REST Kontrolleri	HTTP Rest protokollan mukaisen API:n toteuttava tekninen toiminnallisuus

(taulukko jatkuu)

(taulukko jatkuu)

• Integraatio client	Ulkoisten järjestelmien integraation toteuttava tekninen toiminnallisuus. Ulkoisen järjestelmän tulee toteuttaa rajapinta, johon voidaan integroitua.
• Suodatin (filter)	API-rajapintaan tulevan palvelupyynnön käsittelevä toiminnallisuus. Esimerkiksi pyynnön ja sen lähettämän käyttäjän autentikoinnin ja auktorisoinnin tarkistava toiminnallisuus.
Tietovarastointegraatiokomponentti	Kuvattu luvussa 6.3, taulukossa 16
• Tietovarasto	Tiedon tallennuksen suorittava toiminnallisuus. Käyttää JDBC-protokollaa tietokantaan integroitumiseen.
• Indeksointipalvelu	Tiedon tallennuksen hakuindeksiin suorittava toiminnallisuus. Integroituu hakuindeksiin HTTP Rest-protokollalla.
Tapahtumakomponentti	Kuvattu luvussa 6.3, taulukossa 16
• Tapahtumakäsittelijä	Tietovirralta tulevien tapahtumien käsittelylogiikan sisältävä toiminnallisuus
• Tapahtumakuuntelija	Tietovirran tapahtumia kuunteleva toiminnallisuus. Kuuntelija syöttää tapahtumia tapahtuman mukaiselle tapahtumankäsittelijälle.

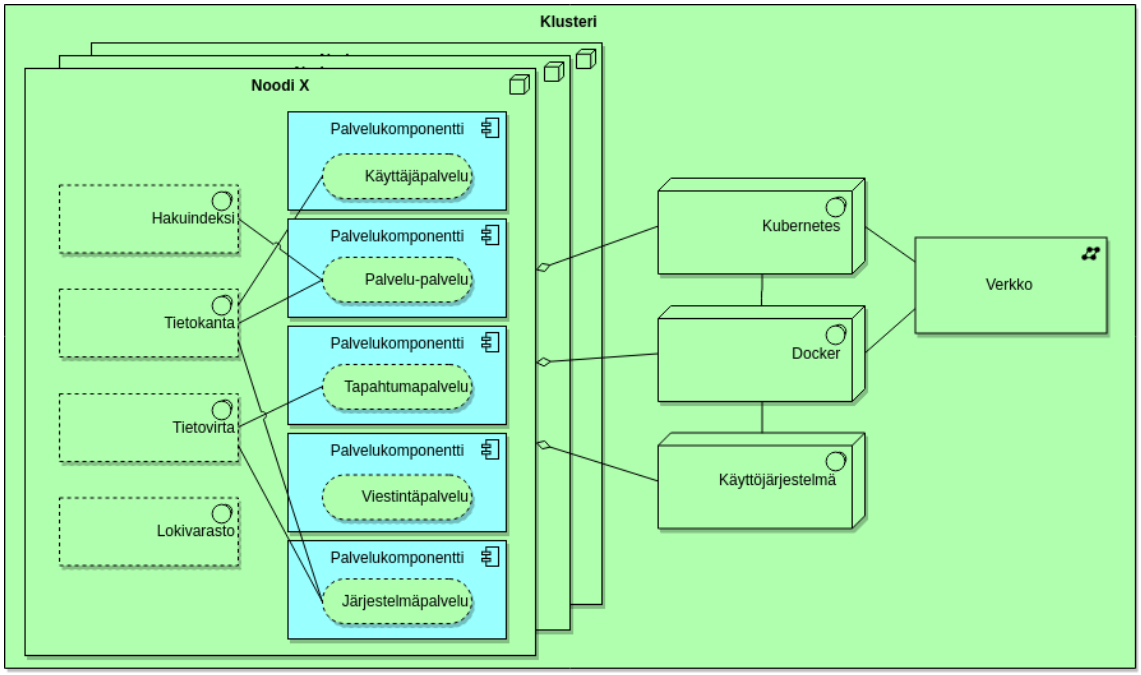
(taulukko jatkuu)

(taulukko jatkuu)

• Tapahtumajulkaisija	Palvelurajapintaan tulevien palvelupyyntöjen käsittelystä seuraavien tapahtumien tietovirtaan tallentava toiminnallisuus
• Integraatio clientti	Tietovirtaintegraation toteuttava tekninen toiminnallisuus. Integroituu tietovirtaan HTTP Rest-protokollalla.
Liiketoimintakomponentti	Kuvattu luvussa 6.3, taulukossa 16
• Liiketoimintakohde	Liiketoiminnallista tietoa ja logiikkaa sisältävä toiminnallisuus. Tietokohde serialisoidaan tietokantaan tallennusta varten (jdbc) ja lähetettäessä API-rajapinnasta vastauksena palvelukutsuun JSON-formaatissa. Deserialisointi tietomalliksi tapahtuu luettaessa tieto tietokannasta ja vastaanotettaessa JSON-muotoinen viesti API-rajapinnasta. Kuvan 16 selvyyden vuoksi yhteydet tietovarastointegraatio ja API-komponenttiin on jätetty piirtämättä.

6.5 Asennusnäkymä

Tässä luvussa esitellään järjestelmän fyysiset komponentit ja aikaisemmissa luvuissa esiteltyjen tasokomponenttien sijoittuminen. Kuvasta 17 nähdään myös järjestelmän toteuttamat fyysiset palvelukomponentti-ilmentymät.



KUVA 17. Järjestelmän arkkitehtuuri, asennusnäkymä

Taulukossa 18 on esitelty kukin kuvan 17 sisältämä komponentti ja sen toiminnallisuus järjestelmässä.

TAULUKKO 18. Asennusnäkymän komponentit

Komponentti	**Kuvaus**
Noodi x	Järjestelmän ajoympäristössä olevien palvelinten kokonaisuus. Palvelinten lukumäärä on joustava. Palvelin voi olla erikoistunut vain tiettyjen arkkitehtuurikomponenttien ajamiseen. Esimerkiksi tietokantapalvelimet ovat vain tietokantaohjelmistojen käyttöön.

(taulukko jatkuu)

(taulukko jatkuu)

	Noodi koostuu käyttöjärjestelmästä, Docker konttien ajoalustasta sekä Kubernetes klusterin ja konttien hallinta järjestelmästä tai sen tarvitsemista agenttitoiminnoista.
Klusteri	Järjestelmä ajoympäristö kokonaisuudessaan. Järjestelmän kehitys-, testaus- ja tuotantoympäristöt toteutetaan omina klustereinaan.
Käyttäjäpalvelu, Palvelupalvelu, Tapahtumapalvelu, Viestintäpalvelu, Järjestelmäpalvelu	Järjestelmän sisältämät palvelukomponenttien toteuttamat mikropalvelut (luku 2.2)
Kubernetes	Klusteriympäristön ja konttien hallintajärjestelmä (luku 3.3)
Docker	Konttien ajoalusta (luku 3.2)
Käyttöjärjestelmä	Noodien käyttöjärjestelmä, käytännössä kehitysympäristössä Linux Ubuntu
Verkko	Fyysinen sekä myös looginen verkko noodien ja konttien välisessä tiedonsiirrossa. Yhteydet noodien ja niiden sisältämien konttien välillä toteuttaa Kubernetes.

7 YHTEENVETO

Tämän opinnäytetyön tarkoituksena oli suunnitella palvelualustan (backend) arkkitehtuuri monikanavaiselle, pääosin mobiilikäyttöiselle uudelle järjestelmälle. Järjestelmä suunniteltiin ja kehitettiin startup-yritykselle sen erityispiirteet ja startup-vaiheen jälkeen tulevat tarpeet huomioiden.

Suunnittelutyön lopputuloksena syntyi suunnitelma arkkitehtuurista, joka vastaa kohteena olevan järjestelmän vaatimuksiin. Vaatimukset jaoteltiin tässä työssä seuraaviin teemoihin:

- startup-yrityksen erityispiirteet
- monikanavaisuus
- monipuolinen toiminnallisuus
- tietoturvallisuus
- suorituskyky, skaalautuvuus
- tekninen alusta.

Arkkitehtuuri ja valitut teknologiat tukevat hyvin ja kustannustehokkaasti startup-yrityksen alkuvaiheen Proof-Of-Concept-tyylistä ketterää tuotekehitystä. Toteutetun järjestelmän testausvaiheessa todettiin myös, että suunnitelman mukainen arkkitehtuuri on käyttökelpoinen ja toimiva.

Mikropalveluiden toteuttaminen yleisiä parhaaksi todettuja käytäntöjä (Convention Over Configuration -paradigma (37)) tukevalla teknologisella viitekehyksellä nopeuttaa standardin mukaisen palvelun toiminnallisuuden toteuttamista. Kypsän viitekehyksen tuki taustapalveluintegraatioille (luku 4.2.6, ”backing services”) nopeuttaa erilaisten palveluiden integrointia ja kokeilemista aiottuun tarpeeseen. Konttiteknologia tuo järjestelmän kehittämiseen myös nopeutta yhdenmukaistamalla sekä kehittäjän oman paikallisen kehitysympäristön, testausympäristön ja tuotannon aikaisen ajoympäristön (luku 4.2.6, ”dev-prod parity”).

Mikropalveluarkkitehtuuri mahdollistaa selkeiden, hyvin rajattujen ja helposti käytettävien API-rajapintojen toteuttamisen. Palveluiden integrointi API-rajapintojen avulla on yksinkertaista mistä tahansa järjestelmän käyttökanavasta (ks. 4.1.2).

Toisaalta mikropalveluarkkitehtuurin ja konttiteknologian palveluihin tuoma toteutusalustariippumattomuus edesauttaa monipuolisen toiminnallisuuden ja integraation kehittämistä, koska toiminnallisuus voidaan toteuttaa sitä parhaiten tukevalla tekniikalla (ks. 2.2).

Nykyaikaiset konttienhallintajärjestelmät, kuten Kubernetes (luku 3.3) ja Rancher (luku 3.4), tuovat arkkitehtuuriin ja ajoympäristöihin hallittavuutta, helppoa skaalautuvuutta ja sitä kautta varmempaa suorituskykyä.

Hyviä käytänteitä ja suunnittelumalleja noudattava mikropalveluarkkitehtuuri tuo haasteita usean mikropalvelun alueelle kohdistuvan transaktion ja sitä kautta tiedon eheyden hallintaan. Tämä haaste ratkaistiin arkkitehtuurissa suunnittelemalla ja toteuttamalla Saaga-mallin (luku 2.3) mukainen tapahtumanhallinta ratkaisu, jossa kahden mikropalvelun välinen tiedoneheys toteutetaan orkestroimalla tiedon päivitystä mikropalveluiden välisillä tapahtumilla tapahtumavirran (luku 3.7) kautta.

Toinen löyhästi toisiinsa integroitujen palveluiden haaste on toimintavarmuus ja virheiden hallinta; virhe tapahtumien käsittelyssä tai tapahtumavirrassa voi potentiaalisesti kadottaa tapahtuman ja sitä kautta tiedon eheys vaarantuu. Tätä ongelmaa ratkaistiin suunnittelemalla ja toteuttamalla kattava tiedon ja tapahtumien keräys keskitettyihin lokeihin.

Tämän opinnäytetyön tekemisen aikana toteutettiin järjestelmän kehitys ja testausympäristö, joten jatkokehittämisessä arkkitehtuurin mukainen tuotantoympäristö tulee rakentaa. Koska arkkitehtuurissa käytetty konttiteknologia paketoi mikropalvelut, tapahtumavirtakomponentin sekä hakuindeksijärjestelmän valmiiksi siirrettäviin komponentteihin, voidaan tuotantoympäristön rakentamisessa keskittyä em. komponenttien asennuksen ja konfiguroinnin sijasta ympäristön skaalautumisen, varmistamisen ja tietoturvan koventamisen tehtäviin.

Toteutetun järjestelmän arkkitehtuuri ei vielä aivan vastaa kaikkia luvussa 2.2 mainittuja mikropalveluarkkitehtuurin piirteitä. Palvelualustan mikropalveluiden automatisoituun testaamiseen sekä automatisoituun jatkuvaan integrointiin ja asennuksiin tulisi vielä kehittää prosesseja ja ottaa käyttöön tarvittavia työkaluja. Myöskään kattavaan monitorointiin ja lokien hallintaan ei ole vielä ollut tarvetta

järjestelmän kehitys ja testausvaiheissa, joten näiden osa-alueiden kehittämiseen tulisi vielä panostaa.

LÄHTEET

1. Li, M. 2017. Lean Startup: “Less is More” and “Fail Fast, Succeed Faster”. Medium Corporation. Saatavissa: https://medium.com/@mianli666/lean-startup-less-is-more-and-fail-fast-succeed-faster-10f3904d4983. Hakupäivä 13.8.2019.

2. Togaf® Standard, Version 9.2. The Open Group. Saatavissa: https://pubs.opengroup.org/architecture/togaf9-doc/arch. Hakupäivä 13.8.2019.

3. The ArchiMate® Enterprise Architecture Modeling Language. 2019. The Open Group. Saatavissa: http://www.opengroup.org/archimate-forum/archimate-overview. Hakupäivä 25.8.2019.

4. Hosiaisluoma, E. 2019. ArchiMate käsikirja. Saatavissa: https://www.hosiaisluoma.fi/blogi/kokonaisarkkitehtuuri/archimate-kasikirja. Hakupäivä 14.8.2019.

5. A Bit History of Internet/Chapter 5: Client-Server. Wikimedia/Wikibooks Open Content Media. Saatavissa: https://en.wikibooks.org/wiki/A_Bit_History_of_Internet/Chapter_5_:_Client-Server. Hakupäivä 14.8.2019.

6. Three-Tier Architecture. Techopedia. https://www.techopedia.com/definition/24649/three-tier-architecture. Hakupäivä 17.9.2019.

7. Devanesan, G. 2018. Layered Architecture in Microservices. DZone, Microservices Zone. Saatavissa: https://dzone.com/articles/layered-architecture-code-smells-medium. Hakupäivä 14.9.2019.

8. Rajput, D. 2017. Create Microservices Architecture Spring Boot. Dinesh on Java. Saatavissa: https://www.dineshonjava.com/microservices-with-spring-boot. Hakupäivä 14.9.2019.

9. Lewis, J - Fowler, M. 2014. Microservices. Saatavissa: https://martinfowler.com/articles/microservices.html. Hakupäivä 14.9.2019.

10. Young, G. 2017. The long sad history of microservices. Video. Saatavissa: https://www.youtube.com/watch?v=MjIfWe6bn40. Hakupäivä 15.9.2019.

11. Kharenko, A. 2015. Monolithic vs. Microservices Architecture. Medium Corporation. Saatavissa: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59. Hakupäivä 14.9.2019.

12. Brown, K. 2018. Beyond buzzwords: A brief history of microservices patterns. IBM Developer. https://developer.ibm.com/articles/cl-evolution-microservices-patterns. Hakupäivä 16.9.2019.

13. Pearlman, S. 2016. Enterprise Service Bus vs Traditional SOA. MuleSoft, LCC. Saatavissa: https://blogs.mulesoft.com/dev/connectivity-dev/esb-vs-soa/. Hakupäivä 21.8.2019.

14. Villamizar, M – Garcés, O – Castro, H – Verano, M – Salamanca, L – Casallas, R – Gil, S. 2015. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. IEEE.

15. Frisk, S. 2019. API-rajapintojen hallinta ja tietoturva. Jyväskylän Yliopisto, Informaatioteknologian tiedekunta. Kandidaatin tutkielma. Saatavissa: https://jyx.jyu.fi/handle/123456789/64334. Hakupäivä 19.10.2019.

16. Richardson, C. 2018. Microservices Pattern chapter 4: Managing transactions with sagas. Manning Publications. Saatavissa: https://livebook.manning.com/book/microservices-patterns/chapter-4. Hakupäivä 15.9.2019.

17. Richardson, C. 2018. Microservices Pattern chapter 6: Developing business logic with event sourcing. Manning Publications. Saatavissa: https://livebook.manning.com/book/microservices-patterns/chapter-6. Hakupäivä 14.9.2019.

18. Niranjan, R. 2018. Imperative, functional, reactive programming – which one to use when and for what? Web Agam. Saatavissa: https://www.webagam.com/2018/11/24/imperative-functional-reactive-programming-which-one-to-use-when-and-for-what. Hakupäivä 27.8.2019.

19. WhatIs.com. Imperative Programming. TechTarget. Saatavissa: https://whatis.techtarget.com/definition/imperative-programming. Hakupäivä 15.9.2019.

20. WhatIs.com. Declarative Programming. TechTarget. Saatavissa: https://searchitoperations.techtarget.com/definition/declarative-programming. Hakupäivä 15.9.2019.

21. Cearley, D. 2017. Top 10 Strategic Technology Trends for 2018. Gartner Inc. Saatavissa: https://www.gartner.com/ngw/globalassets/en/information-technology/documents/top-10-strategic-technology-trends-for-2018.pdf. Hakupäivä 22.8.2019.

22. Swanepoel, H. 2017. A super quick comparison between Kafka and Message Queues. Hackernoon. Saatavissa: https://hackernoon.com/a-super-quick-comparison-between-kafka-and-message-queues-e69742d855a8. Hakupäivä 21.8.2019.

23. Michelson, B.M. 2011. Event-Driven Architecture Overview. Elemental Links, Inc. Saatavissa: http://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf. Hakupäivä 22.8.2019.

24. Tozzi, C. 2017. What Do Containers Have to Do with DevOps, Anyway? MediaOps Inc. Saatavissa: https://containerjournal.com/2017/01/11/containers-devops-anyway. Hakupäivä 21.8.2019.

25. Marquez, E. 2018. The History of Container Technology. Linux Academy. Saatavissa: https://linuxacademy.com/blog/containers/history-of-container-technology. Hakupäivä 24.8.2019.

26. Hardin, T. 2018. Digital Platforms Trends: Containerization. G2.com Inc. Saatavissa: https://learn.g2.com/trends/containerization. Hakupäivä 21.8.2019.

27. Vaughan-Nichols, S.J. 2018. What is Docker and why is it so darn popular? CBS Interactive. Saatavissa: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular. Hakupäivä 21.8.2019.

28. Rhodin, J. 2016. Comparison: Apache Kafka VS RabbitMQ. 84Codes. Saatavissa: https://www.cloudkarafka.com/blog/2016-12-05-apachekafka-vs-rabbitmq.html. Hakupäivä 21.8.2019.

29. Moilanen, R. 2013. Kasvuyritys ja startup-yritys. Kielikello, Kielenhuollon tiedotuslehti / kysyttyä. Saatavissa: https://www.kielikello.fi/-/kasvuyritys-ja-startup-yritys. Hakupäivä 21.8.2019.

30. Blomqvist, H. 2017. Rautakolmio ketterässä projektissa. Oppia.fi. Saatavissa: https://blog.oppia.fi/2017/05/09/rautakolmio-ketterassa-projektissa. Hakupäivä 21.8.2019.

31. Panetta, K. 2016. Why Big Companies Need Lean Startup Techniques. Gartner, Inc. Saatavissa: https://www.gartner.com/smarterwithgartner/why-big-companies-need-lean-startup-techniques. Hakupäivä 21.8.2019.

32. Thomas, A – Gupta, A. 2016. Retire the Three-Tier Application Architecture to Move Toward Digital Business. Gartner Inc. Saatavissa: https://www.gartner.com/binaries/content/assets/events/keywords/applications/apps20i/retire_the_threetier_applica_308298.pdf. Hakupäivä 21.8.2019.

33. Viskari, M. 2012. Monikanavainen e-kaupankäynti. Pro gradu. Tampereen yliopisto, Informaatiotieteiden yksikkö. Saatavissa: http://urn.fi/urn:nbn:fi:uta-1-23180. Hakupäivä 21.8.2019.

34. Hyrkäs, A. 2016. Startup Complexity, Tracing the Conceptual Shift Behind-Disruptive Entrepreneurship. Väitöskirja. University of Helsinki, faculty of social sciences, department of social research. Saatavissa: http://urn.fi/URN:ISBN:978-951-51-2582-8. Haettu 22.8.2019.

35. Wiggings, A. 2017. 12factor Saatavissa: https://12factor.net. Hakupäivä 22.8.2019.

36. Olliffe, G. 2016. Decision Point for Application Services Implementation Architecture. Gartner, Inc. Saatavissa: https://www.gartner.com/en/documents/3371720/decision-point-for-application-services-implementation-a. Haettu 24.8.2019.

37. Convention Over Configuration. Techopedia. Saatavissa: https://www.techopedia.com/definition/27478/convention-over-configuration. Hakupäivä 23.8.2019.

https://github.com/docker-library/mysql/blob/master/5.6/Dockerfile

```
1:  FROM debian:stretch-slim
2:
3:  # add our user and group first to make sure their IDs get assigned consistently,
4:  # regardless of whatever dependencies get added
5:  RUN groupadd -r mysql && useradd -r -g mysql mysql
6:
7:  RUN apt-get update && apt-get install -y --no-install-recommends gnupg dirmngr && rm -rf /var/lib/apt/lists/*
8:
9:  # add gosu for easy step-down from root
10: ENV GOSU_VERSION 1.7
11: RUN set -x \
12:   && apt-get update && apt-get install -y --no-install-recommends ca-certificates wget && rm -rf /var/lib/apt/lists/* \
13:   && wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architecture)" \
14:   && wget -O /usr/local/bin/gosu.asc "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architecture).asc" \
15:   && export GNUPGHOME="$(mktemp -d)" \
16:   && gpg --batch --keyserver ha.pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A9C25BF357DD4 \
17:   && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
18:   && gpgconf --kill all \
19:   && rm -rf "$GNUPGHOME" /usr/local/bin/gosu.asc \
20:   && chmod +x /usr/local/bin/gosu \
21:   && gosu nobody true \
22:   && apt-get purge -y --auto-remove ca-certificates wget
23:
24: RUN mkdir /docker-entrypoint-initdb.d
25:
26: RUN apt-get update && apt-get install -y --no-install-recommends \
27: # for MYSQL_RANDOM_ROOT_PASSWORD
28:                                 pwgen \
29: # for mysql_ssl_rsa_setup
30:                                 openssl \
31: # FATAL ERROR: please install the following Perl modules before executing /usr/local/mysql/scripts/mysql_install_db:
32: # File::Basename
33: # File::Copy
34: # Sys::Hostname
35: # Data::Dumper
36:                                 perl \
37:                 && rm -rf /var/lib/apt/lists/*
```

```
38:
39:  RUN set -ex; \
40:  # gpg: key 5072E1F5: public key "MySQL Release Engineering <mysql-build@oss.oracle.com>" imported
41:    key='A4A9406876FCBD3C456770C88C718D3B5072E1F5'; \
42:    export GNUPGHOME="$(mktemp -d)"; \
43:    gpg --batch --keyserver ha.pool.sks-keyservers.net --recv-keys "$key"; \
44:    gpg --batch --export "$key" > /etc/apt/trusted.gpg.d/mysql.gpg; \
45:    gpgconf --kill all; \
46:    rm -rf "$GNUPGHOME"; \
47:    apt-key list > /dev/null
48:
49:  ENV MYSQL_MAJOR 8.0
50:  ENV MYSQL_VERSION 8.0.15-1debian9
51:
52:  RUN echo "deb http://repo.mysql.com/apt/debian/ stretch mysql-${MYSQL_MAJOR}" > /etc/apt/sources.list.d/mysql.list
53:
54:  # the "/var/lib/mysql" stuff here is because the mysql-server postinst doesn't have an explicit way to disable the mysql_install_db codepath besides having a database already "configured" (ie, stuff in /var/lib/mysql/mysql)
55:  # also, we set debconf keys to make APT a little quieter
56:  RUN { \
57:    echo mysql-community-server mysql-community-server/data-dir select ''; \
58:    echo mysql-community-server mysql-community-server/root-pass password ''; \
59:    echo mysql-community-server mysql-community-server/re-root-pass password ''; \
60:     echo mysql-community-server mysql-community-server/remove-test-db select false; \
61:    } | debconf-set-selections \
62:    && apt-get update && apt-get install -y mysql-community-client="${MYSQL_VERSION}" mysql-community-server-core="${MYSQL_VERSION}" && rm -rf /var/lib/apt/lists/* \
63:    && rm -rf /var/lib/mysql && mkdir -p /var/lib/mysql /var/run/mysqld \
64:    && chown -R mysql:mysql /var/lib/mysql /var/run/mysqld \
65:  # ensure that /var/run/mysqld (used for socket and lock files) is writable regardless of the UID our mysqld instance ends up having at runtime
66:    && chmod 777 /var/run/mysqld
67:
68:  VOLUME /var/lib/mysql
69:  # Config files
70:  COPY config/ /etc/mysql/
71:  COPY docker-entrypoint.sh /usr/local/bin/
72:  RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # backwards compat
73:  ENTRYPOINT ["docker-entrypoint.sh"]
74:
```

```
75:  EXPOSE 3306 33060
76:  CMD ["mysqld"]
```