Harri Kaukovuo

YHDENMUKAINEN TIEDON VALIDOINTI KÄYTTÖ-LIITTYMÄSSÄ JA MIKROPALVELUSSA

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2022

SISÄLLYS

TIIVISTELMÄ

Kaukovuo, Harri
Yhdenmukainen tiedon validointi käyttöliittymässä ja mikropalvelussa
Jyväskylä: Jyväskylän yliopisto, 2021, 72 s.
Tietojärjestelmätiede, Kandidaatintutkielma
Ohjaaja: Kokko, Tuomas

Tässä kandidaatintutkielmassa käydään läpi yhdenmukaisen validoinnin ongelmatiikkaa nykyaikaisen selainpohjaisen käyttöliittymän ja mikropalvelurajapinnan näkökulmasta. Tutkimuskysymyksenä tutkielmassa on: *"Millä tavoilla ja teknologioilla voidaan rakentaa sovellus, jossa samaa validointilogiikkakoodia tai määritystä käytetään niin ohjelmallisten rajapintojen tiedon validoinnissa, kuin käyttöliittymässä?"* Tutkimuksessa käydään ensin läpi nykyaikaisen käyttöliittymän tiedon validointivaihtoehtoja. Tämän jälkeen esitellään JSON-tietorakennetta käyttävän mikropalvelun validointivaihtoehtoja. Vastauksena tutkimuskysymykseen tutkimuksessa esitellään uusi yhdenmukaisen validoinnin ryhmittelymalli toteutustapojen mukaan. Mallin mukaisesti esitellään vaihtoehtoja yhdenmukaisen validoinnin toteuttamiseksi. Tutkimuksen tuloksena todetaan myös, että yhdenmukaista validointia käyttöliittymän ja mikropalvelurajapinnan välillä on toistaiseksi tutkittu vähän. Teknologioiden lukumäärä niin käyttöliittymässä, kuin mikropalveluissa kasvaa vuosittain kehittäjien etsiessä uusia trendikkäämpiä ohjelmistokehikoita, vaikeuttaen yhdenmukaista validointia. Kirjallisuuskatsauksen perusteella näyttää siltä, että alan standardit ovat jäämässä kehityksestä jälkeen, vaikuttaen negatiivisesti yhdenmukaiseen validointiin. Tutkimusmetodina käytettiin tulkitsevaa käsitetutkimusta.

Asiasanat: yhdenmukainen tiedon validointi, javascript, ajax, json validointi, json schema, mikropalvelu, mikropalveluarkkitehtuuri, html, html5

ABSTRACT

Kaukovuo, Harri
Unified data validation in user interface and microservice
Jyväskylä: University of Jyväskylä, 2021, 72 pp.
Bachelor's Thesis, Information Systems
Supervisor: Kokko, Tuomas

Modern web applications and microservices validate the data using different technologies and methods. This Bachelor's Thesis focuses on identifying the data validation methods on modern browser-based front-ends and the validation methods on microservices. This study aims at answering the research question: "*Which methods and technologies are needed to build an application that utilizes the same data validation code or metadata in user interface and application programming interface?*" The study focuses on modern browser technologies and microservices that are built to use JSON documents as the payload. The study presents new model for unified data validation categorization based on the implementation styles. Unified data validation implementation techniques are proposed as part of the categorization model. Study was performed as an interpretative study of concepts. Based on analysis of earlier research, it seems like there are not many studies so far related to this topic. Analysis shows that the number of different technology frameworks is increasing yearly, and popularity of the frameworks is changing based on development trends. Standardization efforts seem to be lagging behind the new innovations causing more issues on unified data validation.

Keywords: unified data validation, javascript, ajax, json validation, json schema, microservice, microservice architecture, html, html5

KUVIOT

TAULUKOT

1 JOHDANTO

Tietojärjestelmien rakentamisen arkkitehtuurit ovat jatkuvassa muutostilassa. Selainpohjaisten RIA-käyttöliittymäteknologioiden kehitys on ollut jatkuvaa, eikä välttämättä aina parempaan suuntaan menetelmällisesti. Selaimessa käytettävien JavaScript-teknologioiden määrä on kasvanut, eikä standardointi ole pysynyt perässä tai standardointi on ollut koordinoimatonta eri standardointiorganisaatioiden välillä (Taivalsaari & Mikkonen, 2011, 2017b).

Taustajärjestelmien muutos monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin on tarkoittanut toisaalta taustajärjestelmien parempia skaalautumismahdollisuuksia ja nopeampaa kehityssykliä, mutta tuonut mukanaan suuren kirjon uusia sovellusteknologioita, joissa kehittäjien aaltoliike kehitysalustasta toiseen muoti-ilmiöiden tapaan on tuonut ongelman, jossa teknologinen ohjelmakoodin uudelleenkäyttö on lähes mahdotonta. Mikropalvelun kehittäminen vain yhtä tarkoitusta varten on tarkoittanut myös sitä, että mikropalvelussa ei välttämättä oteta huomioon isompia suunnittelulinjauksia. Mikropalveluiden ollessa itsenäisiä ja tarjotessa omat itsenäiset rajapinnat käyttöliittymälle ja muille mikropalveluille, olisi tärkeätä huolehtia tietojen validoinnista, tietomallin ja liiketoimintasääntöjen toteuttamisesta jokaisella tasolla ja kaikissa mikropalveluissa. Esimerkiksi tietomallin muutos alimmalla tasolla pitäisi toteuttaa kaikissa tietoa käsittelevissä mikropalveluissa, raporteissa, käyttöliittymissä sekä käyttöliittymäkirjastoissa. Tämä on haasteellista tilanteessa, jossa kehitysryhmät keskittyvät vain omaan palveluunsa. Uusia menetelmiä tarvitaan helpottamaan tiedon validointia, mallien muutosta, käyttöliittymän muutoksia sekä liiketoimintalogiikan muutoksia käyttöliittymien ja mikropalveluiden välillä. (Cerny, T., Donahoo, M. J., & Trnka, 2018.)

Vaikka tiede- ja yritysmaailmassa suunnataan trendinomaisesti kohti mikropalveluarkkitehtuuria, on hyvä huomata, että mikropalveluissa on loppujen lopuksi kyse palveluarkkitehtuurista (Service Oriented Architecture, SOA) ja näin ollen mikropalveluarkkitehtuuri tulee kohtaamaan samoja haasteita kuin mikä tahansa hajautettu järjestelmä: tiedon oikeellisuuden ja johdonmukaisuuden hallinta, palvelurajapintojen suunnittelu ja kehitys sekä sovellushallinta. (Zimmermann, 2017.)

Tekoälyteknologian kehittymisen myötä koneoppimisen kielet ja teknologiat tulevat osaksi järjestelmäarkkitehtuureja. Koneoppimisessa on kyse tietojärjestelmän itseoppimisesta olemassa olevan tiedon, tulevan uuden tiedon ja säännöstön avulla. Koneoppimisessa algoritmit ovat parhaimmillaan silloin kun opetettu tieto on oikein tietotyypitettyä, johdonmukaista ja rakenteellisesti oikein. Validoimaton väärä tieto voi pahimmassa tapauksessa aiheuttaa ennalta arvaamattomia tuloksia koneoppimisen algoritmeissa ja näin ennakoimattomia tuloksia järjestelmän loppukäyttäjälle. Tekoälyteknologioiden kannalta kyse ei ole pelkästään tiedon teknisestä oikeellisuudesta tai loogisesta johdonmukaisuudesta, vaan myös siitä onko koneoppimismalli rakennettu myös oikeudenmukaiseksi, esimerkiksi kaikkia ihmisryhmiä syrjimättömäksi. (Biessmann ym., 2021.) Koneoppimismalleihin ja algoritmeihin perustuva tiedon validointi käyttäen itseoppivaa tekoälyä saattaa olla tulevaisuuden uusi suuntaus.

Tämä kandidaatintutkielma keskittyy kuvaamaan nykyaikaista selainpohjaista mikropalveluarkkitehtuuriin pohjautuvaa sovellusarkkitehtuuria käyttöliittymän ja mikropalvelun näkökulmasta, keskittyen erityisesti tiedon validointiin kummallakin tasolla. Kandidaatintutkielmassa rajataan käyttöliittymän teknologia koskemaan selaimen JavaScript Ajax -toteutuksia JSON-tietorakenteella, koska XML ei ole soveltuvin teknologia Ajax-toteutuksiin (Lin ym., 2012).

1.1 Käsitteet

Seuraavaksi käsitellään tärkeimmät käsitteet, jotka läheisesti liittyvät aiheeseen tai ovat aiheen tutkimisen ja ymmärtämisen kannalta relevantteja. Moni käsite on hyvin abstrakti, jossa kuitenkin tekniset aspektit ja määritteet asettavat tiettyjä reunaehtoja. Tästä johtuen esittelen tarkemmin käytettyjä käsitteitä, teknologiaa ja historiaa, jotta lukija saa yleiskuvan peruskäsitteistä.

1.1.1 RIA-sovellus

Casteleynin, Garrig'osin ja Maz'onin (2014) mukaan termi Rich Internet Application (RIA) esiintyi ensimmäisen kerran vuonna 2002 Macromedia yrityksen faktadokumentissa (engl. white paper), joka kuvasi Macromedia Flash MX -teknologiaa rikkaamman web-teknologian mahdollistajana. Lähes kymmeneen vuoteen termin esittelystä ja vuosikymmenen tutkimusten jälkeen termiä RIA ei oltu määritelty tieteellisesti. Casteley ym. (2014) pyrkivät määrittelemään tutkimuksessaan termin perustuen vuosikymmenen tutkimuksiin ja niiden tuloksiin.

Casteley ym. (2014) mukaan RIA-sovellukset ovat web-sovelluksia, jotka pyrkivät tarjoamaan ominaisuudet ja toiminnallisuudet, joita perinteiset työasemasovellukset tarjoavat. RIA-sovellukset tarjoavat rikkaan, käyttäjää enemmän tyydyttävämmän käyttäjäkokemuksen verrattuna perinteiseen monisivuiseen sovellukseen. Tästä johtopäätöksenä todetaan, että RIA-sovellusten täytyy

pyrkiä parempaan käyttäjävasteeseen, parantamaan käyttäjän vuorovaikutuskykyyksiä ja tarjoamaan käyttäjälle rikkaamman käyttökokemuksen.

RIA-sovellusten toteuttaminen vaatii asynkronista yhteyttä asiakaspäätteen ja palvelimen välillä, tiedon käsittelyn hajauttamista asiakaspäätteen ja palvelimen välillä, sekä rikkaan sovelluslogiikan käyttöliittymäkirjastojen käyttöä asiakaspäätteellä. Rikkaan sovelluslogiikan käyttöliittymäkirjastojen käyttö asiakaspäätteellä mahdollistaa esimerkiksi selainsovelluksen käyttöliittymän toteuttamisen niin, että se vastaa lähes alkuperäisen työasemasovelluksen käyttöliittymäkokemusta. (Casteleyn ym., 2014.)

Asynkroninen yhteys asiakaspäätteen ja palvelimen välillä on avain turhan tiedonsiirron välttämiselle, mahdollistaen verkkoliikenteen minimoimisen ja käyttöliittymän tarvittavien osien päivittämiselle. Asynkronisen teknologian käyttö mahdollistaa vain tarpeellisen tiedon välittämisen selainsovelluksen sivujen päivittämisessä. (Casteleyn ym., 2014.) Perinteisissä selainsovelluksissa, joissa koko selainsivu päivitetään, on todettu jopa 57 %:ia tarpeettoman tiedon välittämistä. Tarpeettomalla tiedolla tarkoitetaan tietoa, joka on ollut jo selaimen näytöllä, mutta joka haetaan uudelleen tiedon päivittämisen yhteydessä (Bouras & Konidaris, 2005).

Tiedon ja tiedonkäsittelyn hajauttaminen asiakaspäätteen ja palvelimen välille tarkoittaa asiakassovelluksen käyttämistä tiedon tallennuksessa ja tiedon käsittelyssä. Perinteisissä monisivuisissa verkkosovelluksissa tiedon tallennus ja käsittely on tehty pääosin palvelimessa. Tiedon käsittely asiakasselaimessa vaatii selainpohjaisia tiedon tallennusratkaisuja ja operointilogiikan hajauttamista asiakassovellukseen esimerkiksi JavaScript-kirjastoilla. Tiedon käsittelyn ja tallennuksen hajauttaminen vähentää tarpeettomia palvelinkutsuja. (Casteleyn ym., 2014.) Esimerkkejä selaimessa käytettävistä tallennusteknologioista ovat muun muassa Flash LSO, Google Gears ja HTML5-tietokanta (Zhao ym., 2010).

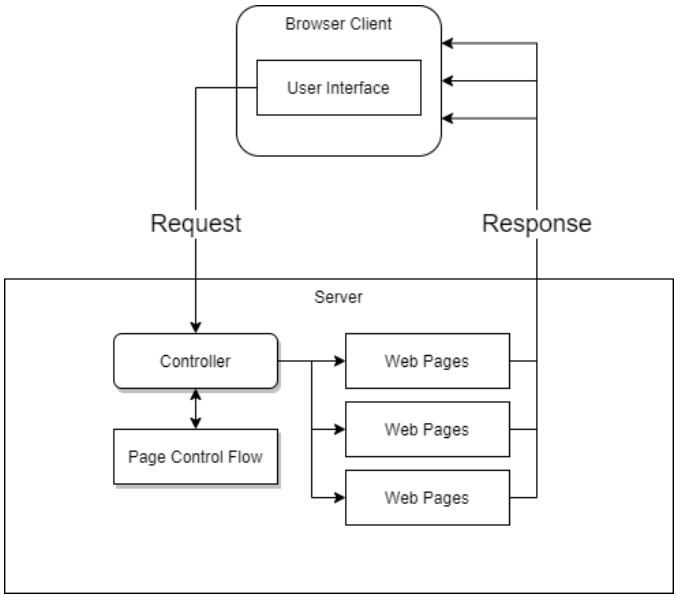
Huomattakoon, että RIA termistä ja sen sisältämästä teknologiasta on erilaisia näkemyksiä. Esimerkiksi Taivalsaari ja Mikkonen (2017) määrittelevät RIA-sovellusten olevan 2000 luvun lopun web-sovelluslaajennusten, kuten Adobe AIR tai Microsoft Silverlight, tapaisia teknologioita. Casteley ym.(2014) Busch ja Koch sekä Fratenali (2009), Rossi ja Sanchez-Figueroa (2010) määrittelevät RIA-termin yleisemmäksi (Busch & Koch, 2009; Fraternali ym., 2010). Tässä tutkimuksessa käytetään yleisempää määritystä RIA-termistä.

1.1.2 Monisivusovellus ja yksisivusovellus

Verkkoselainten historian alkuajoista on selainsovellusten rakennusmalli perustunut malliin, jossa palvelin käsittelee tiedot ja tuottaa selaimelle valmiin HTML-sivun, joka näytetään selaimessa. Tätä mallia on kutsuttu klassiseksi verkkosovellusmalliksi. Mallissa sovelluslogiikka jakaantuu useammalle websivulle, muodostaen sovelluksen, jota voidaan kutsua myös monisivusovellukseksi (multi-page application, MPA). (Mesbah & van Deursen, 2007.)

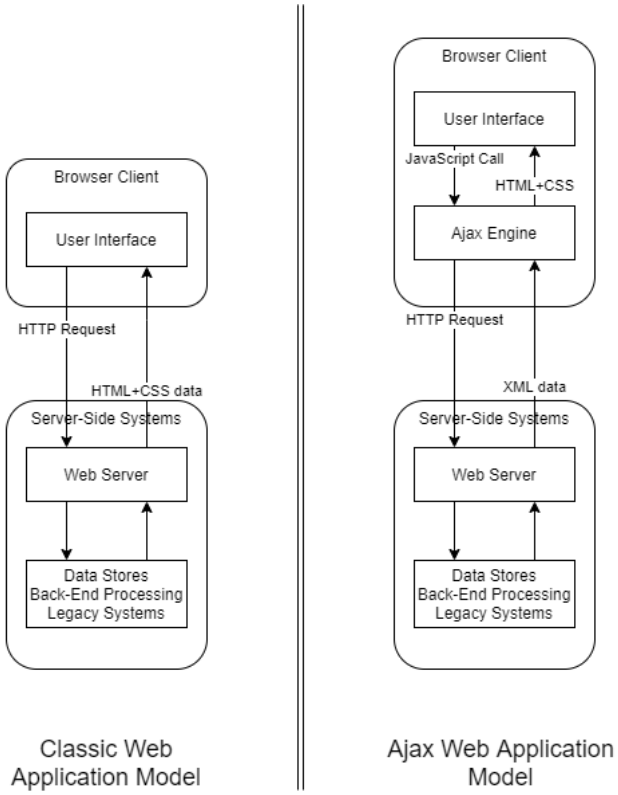
Ping, Kontogiannis ja Lau (2003) kuvaavat tutkimuksessaan monisivuselainsovellusten konvertoimista Model-View-Controller (MVC) arkkitehtuuriin.

MVC-arkkitehtuurissa sovelluksen sivusiirtymälogiikka kuvataan kontrollerimoduulissa. Monisivusovelluksissa tämä siirtymälogiikka on yleensä palvelimessa, tarkoittaen sivun uudelleen latausta palvelimesta siirryttäessä sivulta toiselle. (Ping ym., 2003.) Monisivusovelluksen MVC-arkkitehtuurissa sovelluksen liiketoimintalogiikka on yleensä rakennettu palvelimelle kontrolleriluokkaan. Tämän kandidaattityön kannalta tässä on yksi merkittävä ero seuraavaksi esiteltävään yksisivusovellukseen: monisivusovelluksessa palvelimen MVC-arkkitehtuurin mukaiseen malli- tai kontrolleriluokkaan voidaan helpommin rakentaa keskitetty tietojen validointi, koska näyttörakentaminen tehdään palvelimessa.



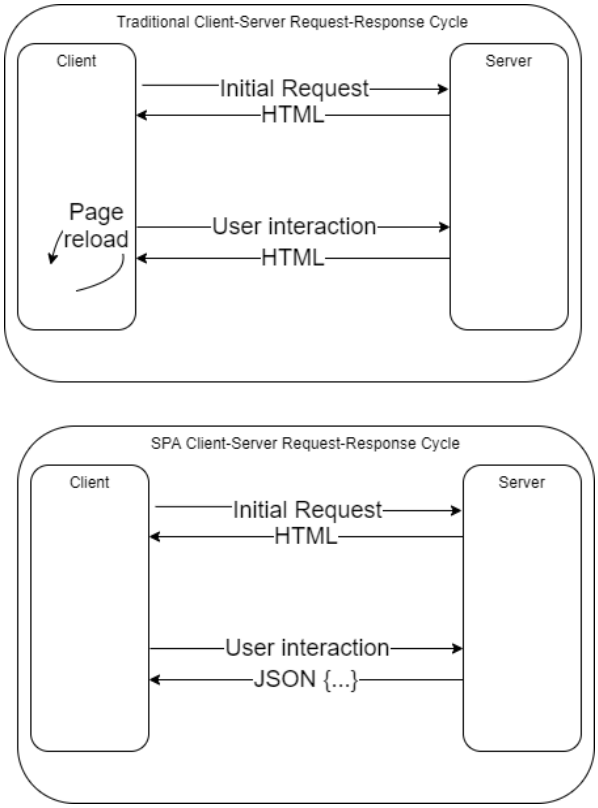
KUVIO 1 Monisivusovelluksen kontrollerikeskeinen arkkitehtuuri (Ping ym., 2003)

Adaptive Path -yrityksen J.Garrett (2005) esitteli termin Asynchronous JavaScript + XML (Ajax), jolla tarkoitettiin teknologioita, joilla mahdollistetaan selaimen sovelluksen monimuotoinen ja rikas toiminnallisuus samaan tapaan kuin työpöytäsovelluksissa. Verkkosovellus rakennettaisiin yksisivuiseksi, ja tiedonsiirto taustajärjestelmiin tapahtuisi dynaamisesti ja taustatoiminnallisuuksina ilman, että käyttäjän täytyy odottaa verkkosivujen täydellistä latautumista. (Garrett, 2005.) Yksisivuista sovellusta kutsutaan englanninkielisellä termillä Single-Page Application (SPA).



KUVIO 2 Klassinen verkkosovellusmalli verrattuna Ajax-malliin (Garrett, 2005)

Modernit web-sovellukset rakennetaan nykypäivänä usein SPA-sovelluksina, jossa yksi web-sivu toimii sovelluksen pohjana, ja sivun päivitykset tapahtuvat dynaamisesti ilman koko web-sivun uudelleen lataamista. Moderneja web-sovellusten JavaScript-ohjelmointikehikoita on lukuisia, joista yksi suosituimmista on esimerkiksi AngularJS, joka on Googlen ylläpitämä (Jadhav ym., 2015). AngularJS sisältää selaimessa ajettavan Model-View-Controller (MVC) -arkkitehtuurin mukaisen ohjelmointikehikon. MVC-kehikko mahdollistaa SPA-sovelluksessa esimerkiksi käyttäjän navigointihistorian säilyttämisen MPA-sovelluksen tapaan (Pinto & Coutinho, 2018). SPA-sovelluskehityksen yksi haasteista on modernien Javascript-pohjaisten ohjelmistokehikoiden lukuisa määrä ja kehikoiden keskinäinen kilpailu kehittäjien keskuudessa. Esimerkiksi AngularJS tilalle on nousemassa muita teknologioita kuten React.js. (Taivalsaari & Mikkonen, 2017b.)



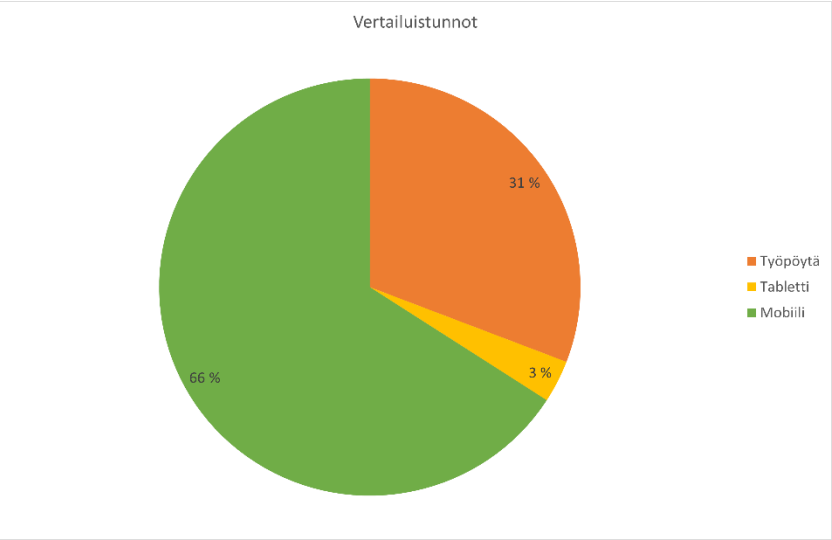
KUVIO 3 Client-Server Request-Response Cycle (Jadhav ym., 2015)

Yu ja Kong (2016) ovat todenneet tutkimuksessaan, että mobiilikäyttäjät pitävät SPA sovelluksia helpompikäyttöisinä. Mobiilikäyttäjät pääsääntöisesti navigoivat liu'uttamalla näyttöä oikealta vasemmalle tai päinvastoin, navigoidessaan seuraavalle sivulle. Myös listojen selaaminen tapahtuu rullaamalla näyttöä alas ja ylös, sen sijaan että listoja selattaisiin web-sivu kerrallaan (MPA-tyylillä). Suurin osa tutkituista kansainvälisistä uutissivustoista on rakentanut mobiiliversionsa SPA-versiona. (Yu & Kong, 2016.)

Bröhl, Rasche, Jablonski, Theis, Will ja Mertens (2018) ovat todenneet, että mobiililaitteiden (sisältäen älypuhelimen ja tabletin) käyttö on ohittanut työpöytä- tai kannettavan tietokoneen käytön. Samaan trendiin viittaavat myös Pinto ja Coutinho (2018) viitatessaan ladattujen sovellusten määrään verrattuna työpöytä- ja mobiililaitteita toisiinsa.

Tässä kandidaatintutkielmassa esitetään aiempia tutkimustuloksia vahvistava tilastollinen kuva, joka perustuu kirjoittajan oman Google Analytics -tilin mahdollistamaan sivustovertailuun verrattuna globaaliin analytiikkaan halutulta ajanjaksolta (1.1.2021—20.11.2021). Globaali analytiikka saadaan esille valitsemalla maantieteellinen rajaus, toimiala, istuntojen määrä per päivä ja haluttu aikajakso. Analyysiin valittiin globaalit sivustot, jotka kuuluvat "Ostokset" (muun muassa verkkokaupat ja huutokaupat) toimialalle, joissa on 100—499 istuntoa päivittäin ja aikajaksolta 1.1.2021—20.11.2021. Tuloksena raportti, jossa oli analysoitu 20558 verkkosivustoa. Lopputulos tukee aiempia tutkimuksia, osoittaen tämän analyysin mukaan työpöytäkäyttäjien määrän olevan 31 % ja mobiilikäyttäjien määrän 69 %. Mobiilikäyttäjien määrään on laskettu mukaan

älypuhelin- ja tablet-käyttäjät. Mobiilikäyttäjien määrän kasvaessa paine yksisivusovellusten kehittämiselle kasvaa.



KUVIO 4 Työpöytä, tabletti ja mobiilikäyttäjien istuntojen jakauma 1.1.2021–20.11.2011

Mobiilikäyttäjien määrän kasvaessa mielenkiinto kohdistuu myös sovelluskehityskehikoihin, joilla mobiililaitteissa toimivia sovelluksia kehitetään. Pinto ym. (2018) kuvaavat tutkimuksessaan tutkimusyhteisössä yleisesti käytettyä luokittelua: natiivi, selainpohjainen ja hybridi kehitystapa. Natiivitavalla mobiililaitteessa käytetään laitteen omaa ohjelmointikieltä ja sen rajapintoja. Selainpohjaisessa kehityksessä mobiililaite käyttää sovellusta verkon yli laitteen selaimen avulla. Hybridikehityksessä yhdistetään nämä kaksi, natiivisovelluksen sisällä ajetaan WebView -tekniikalla HTML5-pohjaista sovellusta. Näistä selainpohjainen ja hybriditoteutukset ovat nopealla tahdilla saavuttamassa natiivisovellusten toiminnallisuutta ja suorituskykytasoa, samalla mahdollistaen monen alustan kehityksen samalla ohjelmakoodilla. (Pinto & Coutinho, 2018.) Erityisen kiintoisaa tämä kehitys on tämän kandidaatintyön aihealueen kannalta, koska HTML5-pohjainen kehitys mahdollistaa työpöytäselaimen ja mobiilisovellusten validointilogiikan yhdenmukaistamisen. Tässä tutkimuksessa keskitytään teknologioihin ja web sovelluksen arkkitehtuureihin, joilla rakennetaan SPA sovelluksia.

1.1.3 JSON

JavaScript Object Notation (JSON) tiedonsiirtoformaatti esiteltiin julkisesti json.org web-sivustolla vuonna 2001. JSON-tiedonsiirtoformaatin ensimmäinen standardointiversio julkaistiin IETF-organisaatiolle ehdotuksena RFC 4627 heinäkuussa vuonna 2006. JSON-tietomuoto on yksinkertainen ja kevyt tiedon esittämistapa. (ECMA International, 2017.)

Esimerkki JSON-dokumentista:

```
{
  "glossary": {
```

```
        "title": "example glossary",
        "GlossDiv": {
          "title": "S",
          "GlossList": {
            "GlossEntry": {
              "ID": "SGML",
              "SortAs": "SGML",
              "GlossTerm": "Standard Generalized Markup Language",
              "Acronym": "SGML",
              "Abbrev": "ISO 8879:1986",
              "GlossDef": {
                "para": "A meta-markup language, used to create markup
languages such as DocBook.",
                "GlossSeeAlso": ["GML", "XML"]
              },
              "GlossSee": "markup"
            }
          }
        }
      }
    }
```

1.1.4 Ajax

Garrett (2005) esitteli termin ”Asynchronous JavaScript + XML (Ajax)” yrityksensä Adaptive Path teknisessä artikkelissa. Ajax ei ole yksittäinen teknologia, vaan useita teknologioita, jotka koostuvat

- standardipohjaisista esitystavoista: XTHML ja CSS
- dynaamisesta näytön esittämisestä ja käyttäjävuorovaikutuksesta pohjautuen selaimen Document Object Model (DOM) -ohjelmointimalliin
- tiedon siirrosta ja käsittelystä XML- ja XSLT-teknologioilla
- taustalla tapahtuvasta asynkronisen tiedon hausta käyttäen XMLHttpRequest selainkutsuja
- JavaScript-ohjelmoinnista, jolla kaikki edellä mainitut sidotaan yhteen. (Garrett, 2005).

Myöhemmin on JSON-tietoformaatti syrjäyttänyt XML-tietoformaatin helppokäyttöisyytensä ja suorituskykyisyytensä ansiosta (Breje ym., 2018; Lin ym., 2012; Simec & Maglicic, 2014). Termiä Ajax käytetään yleisesti siitäkin huolimatta, että XML-formaattia ei valtaosin enää käytetä.

1.1.5 Tiedon validointi

Yleisesti hyväksyttyä tieteellistä tapaa tiedon oikeellisuustarkistuksen metodien ja prosessien määrittelyyn ei ole muodostettu. Tähän tutkimukseen otetaan tiedon validoinnin määrittely vuonna 2014 aloitetusta työstä, jota Euroopan Unionin tilastoviranomainen (Eurostat) rahoitti osana ESSNet projektia ”Harmonizing data validation approaches in the ESS (ESSNet ValiDat - Foundation)” (Karlberg, 2015). European Statistical System (ESS) on yhteenliittymä, jossa

kumppaneina ovat Eurostat, kansalliset tilastokeskukset (National Statistical Institutes, NSIs) ja muut kansalliset viranomaiset (ESS, 2021).

Edellä mainitusta projektista lopputuotoksena oli ”Methodology for Data Validation 1.0 Rev. Jun 2016” (Di Zio ym., 2016). Tässä tutkimuksessa viitataan uudempaan versioon 2.0, joka on julkaistu kaksi vuotta myöhemmin (ESS, 2018).

ESS (2018) käsikirjassa ”Methodology for data validation” määritellään tiedon validointi seuraavasti: *Tiedon validointi on toiminto, jossa todennetaan, onko vai ei yhdistelmä tietoa osa hyväksyttävää yhdistelmää tietoa*. Tällä määritelmällä pyritään esittämään se, että validoinnin tehtävä on myös informoida virheellisestä tiedosta. Yhdistelmä tietoa voi olla yksittäinen kenttä tai se voi olla kenttä yhdistettynä isompaa joukkoa, kuten tietue, sarake tai laajempi tietojoukko.

ESS metodologiassa validointisäännöt jaetaan karkeasti kahteen ryhmään: rakenteelliseen validointiin ja sisällölliseen validointiin. Rakenteellisella validoinnilla tarkoitetaan teknistä sisällön tarkistusta rakenteen ja formaattien kannalta. Tähän kategoriaan kuuluu tietoarvojen rakenteellinen tarkistus, esimerkiksi tietotyyppi, kentän pituus ja numeroarvorajat. Lisäksi arvojen olemassaolon tarkistus, duplikaattien tarkistus, tarkistus koodilistoja vasten sekä hierarkkisen rakenteen tarkistus. Rakenteellisen validoinnin esimerkkejä ovat päivämääräformaatin tarkistus, sosiaaliturvatunnuksen oikeellisuuden tarkistus ja puuttuvan osoitetiedon tarkistus.

Sisällöllisessä validoinnissa keskitytään loogiseen validointiin ja sisällön johdonmukaisuuteen. Säännöt ovat myös usein ehdollisia, eli tiedon validiteetti tarkistetaan, jos tietyt ehdot täyttyvät. Sisällöllisen validoinnin esimerkkejä ovat: ”jos ikä alle 15 vuotta, niin aviosääty ei voi olla naimisissa” tai ”jos työntekijöiden määrä on suurempi kuin nolla, palkkojen täytyy olla suurempi kuin nolla” tai ”työllistettyjen määrä täytyy olla työntekijöiden ja yksinyrittäjien summa”. (ESS, 2018.)

ESS metodologiassa on keskitytty tilastollisen tiedon validointiin ennen sen käsittelyä tilastotietojen analysoinnissa. Tämä metodologia tarjoaa kuitenkin hyvän pohjan myös yleisemmälle tiedon validoinnin määrittelylle. Metodologiaa ei voine sellaisenaan käyttää reaaliaikaisen järjestelmän tietojen validointiin, mutta sieltä voidaan käyttää hyväksi metodologiassa kehitettyä tiedon validoinnin tasomääritystä liiketoiminnallisia näkökulmasta. Metodologia käyttää termiä ”domain”, jolla ESS (2018) tarkoitti Euroopan Unionin eri tilastollisia kokonaisuuksia, mutta joita voitaisiin tässä tutkimuksessa tulkita järjestelmäkokonaisuudeksi. Alla olevassa taulukossa on kuvattu ESS metodologian validointitasot. Kuvaus -sarakkeen validointikuvausta on muutettu yleisemmäksi, jolloin se soveltuu myös tämän kandidaatintutkielman aihepiiriin.

TAULUKKO 1 Tiedon validoinnin tasot liiketoiminnallisesta näkökulmasta (ESS, 2018)

Validointitaso	Kuvaus	Validointiryhmä
1	Rakennetietojen tarkistus tiedosto- tai rajapintakutsutasolla.	Rakennetarkistus
2	Sisältötietojen tarkistus eri tiedostojen tai eri rajapintakutsujen/mikropalvelujen tasolla.	Sisältötarkistus

3	Sisältötietojen tarkistus samaan järjestelmäkokonaisuuteen kuuluvien tietolähteiden kanssa.	Sisältötarkistus
4	Sisältötietojen tarkistus saman organisaation sisällä eri järjestelmäkokonaisuuteen kuuluvien tietolähteiden kanssa. Esimerkki: Organisaation tullausjärjestelmän tiedot täytyy täsmätä satamaoperaattoreiden tietojen kanssa.	Sisältötarkistus
5	Sisältötietojen tarkistus eri organisaatioiden tai maiden tietojärjestelmien kanssa. Esimerkki: Rajapintakutsut esimerkiksi ulkomaisten yritysten VAT ID tiedon varmistamiseksi.	Sisältötarkistus

1.2 Tutkimusmetodi

Tutkimusaiheen alustavaa tutkimusmateriaalia analysoitaessa kävi varsin pian selväksi, että tiedon validoinnin käsitteistö ja metodologiat olivat varsin hajanaisia. Näkökulmia aihealueesta löytyi useita, jokaisella oma rajautunut katsanto tiedon validointiin. Tässä tutkimuksessa tutkimusmetodina käytetään tulkitsevaa käsitetutkimusta kirjallisuuskatsauksen apuna (Takala & Lamsa, 2001). Kandidaatintutkielmassa käytettiin vahvistavana metodina kevyttä empiiristä osuutta. Empiriaa käytettiin tarkistamaan tämän hetken tilannetta, koska osa tutkimuksista oli vuosia vanhoja. Näitä empiirisiä osia olivat haut StateofJS sivuston kyselyanalytiikkakoneesta, Stack Overflow sivuston kehityskysymysten analytiikkaa vuosilta 2009—2021, otos Google Analytics globaaliin vertailuaineistoon tarkistettaessa työpöytäkäyttäjien määrää mobiilikäyttäjiin sekä relaatio- ja NoSQL-tietokantojen JSON-validoinnin vertailu. Empiirisiä osuuksia käytettiin vahvistamaan ja todentamaan aiempien tutkimuksien löydöksiä. Kandidaatintutkielmassa viitataan samoihin analytiikkalähteisiin, joita tutkimuskirjallisuudessa oli käytetty.

Kirjallisuuskatsauksen pääasiallisena hakukoneena käytettiin Google Scholar -hakukonetta. Saaduista hakutuloksista pyrittiin valitsemaan ne tutkimukset, jotka ovat saaneet mahdollisimman monia tutkimusviittauksia ja suosittiin niitä, jotka oli julkaistu JUFO 2 tai 3 tason julkaisuissa. Tutkimusviittauksiin suhtauduttiin kuitenkin kriittisesti. Tuloksista löytyi useampia tutkimuksia, joihin oli esimerkiksi useita kymmeniä viittauksia, mutta tutkimus itsessään vaikutti hyvin pinnalliselta ja esimerkiksi sisälsi tarkistettaessa vääriä lähdeviitteitä, suoranaisia kopiovirheitä (kuvissa kopioitu laatikoita, joissa jäänyt sama teksti) tai tarkistettaessa lähdeviitteissä ei ollut lainkaan mainintaa viitattuun asiaan. Joissakin tilanteissa tähän kandidaatintutkielmaan otettiin vähemmän viitteitä saatuja tutkimusviittauksia, koska itse tutkimus sisälsi oivaltavia ja aihealueeseen liittyviä ideoita.

Hakusanoja, joita käytettiin haettaessa, oli kymmeniä aihealueen vakiintumattoman termistön vuoksi. Esimerkkejä hakusanoista ovat: ”form validation” ”validation”, ”RIA”, ”API”, ”MPA”, ”SPA”, ”Multi-page

interaction", "single-page interaction", "JSON", "Modern web application", "MVC", "Microservices", "Microservices architecture", "AngularJS", "vuejs", "react", "redux", "JSON validation", "JSON schema", "unified data validation", "data validation methodology", "ESS, "schema validation", "rule engine", "inference", "RuleML", "SWRL", "OWL", "JSON-LD" ja "cross-platform development".

1.3 Tutkimusongelma

Tämän tutkielman tarkoituksena on selvittää, kuinka validointimekanismeja voidaan hyödyntää käyttöliittymässä ja mikropalvelurajapinnoissa. Tämän seurauksena muodostui aihepiiriä kuvaava ja ilmentävä tutkimusongelma: Millä tavoilla ja teknologioilla voidaan rakentaa sovellus, jossa samaa validointilogiikkakoodia tai määritystä käytetään niin ohjelmallisten rajapintojen tiedon validoinnissa, kuin käyttöliittymässä?

1.4 Tarkennus ja rajaus

Tutkimus rajataan käsittämään vain nykyaikaisen selainpohjaisen Rich Internet Application (RIA) -käyttöliittymän näkökannalta, taustajärjestelmän rajapintojen ollessa HTTP JSON -pohjaisia mikropalveluarkkitehtuurin perustuvia palveluita.

Lisäksi kartoitetaan, kuinka selainkäyttöliittymässä käytettäisiin standardoituja ja selaimissa mukana tulevia teknologioita, jotka tukevat JSON-tietojen käsittelyä. Esimerkiksi XForms, joka pohjautuu XML-teknologiaan, jätetään vertailusta pois JSON-rajauksen takia ja myös siksi, että tämän teknologian tuki on käytännössä jäänyt pois selaimista vuosia sitten.

Tutkimus jakaantuu kolmeen sisältölukuun, joissa kartoitetaan ja tehdään näkyväksi niitä ongelmia ja haasteita, joita teemaan liittyy. Ensimmäisessä sisältöluvussa käydään läpi selaimessa ajettavan sovelluksen tietojen validointivaihtoehtoja. Toisessa luvussa keskitytään mikropalveluarkkitehtuuriin, ensin perustellen miksi mikropalveluarkkitehtuuri on tällä hetkellä yksi keskeisimmistä järjestelmäarkkitehtuurisuuntauksista, jatkuen JSON-tietorakenteen validointitapoihin. Kolmannessa sisältöluvussa kuvataan yhdenmukaisen tiedon validoinnin toteutusmetodeja, ajatuksena esitellä ne tavat, joilla käyttöliittymässä ja mikropalveluissa voitaisiin käyttää mahdollisimman samaa validointilogiikkaa. Yhteenvedossa kuvataan aihealueen yleistä tutkimuksen tilaa tällä hetkellä, kuvataan selainpohjaisen validoinnin päätavat, mikropalvelujen tiedon validoinnin päätavat ja tehdään kooste yhdenmukaisen tiedon validoinnin päätavoista. Yhteenvedossa myös ehdotetaan jatkotutkimusaiheita.

2 SELAIMESSA AJETTAVAN YKSISIVUSOVELLUKSEN TIETOJEN VALIDOINTI

Tässä luvussa käydään läpi selaimessa ajettavan yksisivusovelluksen käyttöliittymän tietojen validointilogiikkaa päätavoillaan. Aluksi käydään läpi selaimessa käytettävän HTML standardoinnin historiaa, jolla perustellaan HTML-standardikokonaisuus, jonka validointilogiikkaa tarkastellaan tarkemmin.

HTML historia ja HTML-standardin validoinnin yksityiskohdat perustuvat suoraan standardiin ja näin ollen yksioikeutettuna lähteenä on pidetty WHATWG-organisaation HTML-standardia (WHATWG, 2021). WHATWG-organisaation tausta on kuvattu tarkemmin HTML historia -kappaleessa.

2.1 HTML historia

WHATWG (2021) mukaan selaimessa käytettävä Hyper Text Markup Language (HTML) historia alkaa vuosista 1990–1995, jolloin HTML kävi läpi useita versiomuutoksia, pääsääntöisesti CERN (European Organization for Nuclear Research) organisaation kehitysprojekteina, joissa brittitutkija Tim Berners-Lee keksi ja kehitti World Wide Web (WWW) tutkimuksellisiin tiedonjaon tarpeisiin. Myöhemmin HTML julkistettiin IETF standardiksi, jossa sitä standardointiin vuoteen 1994 asti. World Wide Web Consortium (W3C) perustettiin vuonna 1994 ja HTML kehitys jatkui tässä organisaatiossa. HTML 3.0 julkistettiin 1995, HTML 3.2 viimeisteltiin 1997 ja HTML4 seurasi nopeasti samana vuonna 1997. (WHATWG, 2021.)

HTML4:än jälkeen vuonna 1998 W3C päätti lopettaa HTML:än kehittämisen ja keskittyä XML-pohjaisen vastaavan kuvauskielen kehittämiseen. Tätä kutsuttiin XHTML:ksi. XHTML versio 1.0 saatiin valmiiksi vuonna 2000, mutta tämä versio ei sisältänyt uusia ominaisuuksia, ainoastaan uuden tavan sarjallistaa (engl. serialize) HTML-dokumentteja. Yhtä aikaa XHTML-kehittämisen kanssa W3C työskenteli uuden XHTML2-kuvauskielen parissa. XHTML2 ei ollut yhteensopiva aiempien HTML:än tai XHTML:än kanssa. (WHATWG, 2021.)

Vuonna 2003 XForms-teknologian julkaiseminen aiheutti uuden kiinnostuksen jo W3C hylkäämään HTML-kuvauskieleen. XForms-teknologia mahdollisti HTML-yhteensopivien selainmoottoreiden hyödyntämisen. Ideasta järjestettiin vuonna 2004 W3C työpaja, jossa keskusteltiin HTML4-standardin uudelleen avaamisesta jatkokehitykselle. Idean esittivät Mozilla- ja Opera-organisaatiot W3C konsortiolle, joka äänestyksen jälkeen päätti hylätä HTML4:än jatkokehittämisen sen takia, että päätös olisi merkinnyt ristiriitaa aiemman XML-pohjaisen XHTML2:en kehityksen kanssa. W3C päätti jatkaa XML-pohjaisen XHTML2:en kehittämistä HTML:än tilalle. (WHATWG, 2021.)

Pian W3C-työpajan jälkeen Apple, Mozilla ja Opera yhdessä ilmoittivat jatkavansa HTML-pohjaisen standardin kehittämistä uuden yhteistyöorganisaation Web Hypertext Application Technology Working Group (WHATWG) sisällä. Nykyisin organisaation ohjausryhmään kuuluvat myös Google ja Microsoft. WHATWG organisaation perusperiaatteisiin kuuluvat: tietyt teknologiat pitää olla taaksepäin yhteensopivia, määritykset ja toteutukset täytyy olla yhdenmukaisia ja määritykset täytyy olla niin yksityiskohtaisesti kuvattu, että toimittajien selaintoiminnallisuudet voidaan toteuttaa ilman tarvetta takaisinmallintaa kilpailijoiden toteutuksia yhteensopivuuden saavuttamiseksi. (WHATWG, 2021.)

HTML5-standardin kehittämiseksi siihen liitettiin kolme aiemmin erillisinä pidettyä standardointidokumenttia: HTML4, XHTML1 ja DOM2 HTML. Myös HTML5-standardin kuvausta tarkennettiin huomattavasti aiemmasta paremman yhteensopivuuden saavuttamiseksi. Vuonna 2006 W3C ilmoitti halukkuutensa osallistua HTML5 kehitystyöhön, huolimatta aiemmasta kielteisestä päätöksestään. Seuraavana vuonna 2007, W3C ja WHATWG muodostivat työryhmän uuden HTML5-standardin kehittämiseksi. Useita vuosia nämä kaksi työryhmää työskentelivät yhdessä. Vuonna 2011 ryhmät päätyivät johtopäätökseen, että näillä kahdella ryhmittymällä oli toisistaan eriävät päämäärät. W3C halusi julkaista valmiin HTML5-standardin, kun taas WHATWG halusi työstää jatkuvasti kehityksen alla olevaa HTML-standardia (”Living Standard for HTML”). Vuonna 2019 WHATWG ja W3C ilmoittivat allekirjoittaneensa yhteistyösopimuksen, jonka mukaan kumpikin organisaatio kehittää yhtä yhteistä versiota HTML-standardista. Tätä standardia kutsutaan termillä ”HTML Living Standard”. (WHATWG, 2021.)

Tässä tutkimuksessa HTML-standardilla viitataan tähän parhaillaan kehityksen alla olevaan HTML Living Standard -mukaiseen määritykseen, jonka päivämäärä on 9.11.2021. Koska kyseessä on jatkuvasti päivittyvä standardi, voi olla, että myöhemmin luettuna tämän tutkimuksen olettamukset eivät pidä paikkaansa sen hetken standardin määrityksiin nähden. Tutkimuksissa ja internetistä löytyvissä artikkeleissa yleisesti viitataan termiin HTML5, joka ei varsinaisesti tarkoita HTML5-standardia, koska sellaista ei virallisesti ole olemassa. Tällä yleisesti tarkoitetaan HTML4:än jälkeistä HTML-määritystä, johon viitataan virallisesti nimellä ”HTML Living Standard”.

2.2 HTML-sivujen validointitavat

HTML-standardissa määritellään, että HTML-lomakkeiden tiedot voidaan validoida selaimessa ennen kuin tiedot lähetetään palvelimelle. Palvelimen on kuitenkin validoitava tiedot uudelleen, koska selaimen tietojen validointi voidaan helposti ohittaa. HTML lomakkeiden validointitapoja on kaksi. Käyttäen HTML-standardissa määriteltyjä syöttökenttien validointiominaisuuksia tai käyttäen JavaScriptiä. (WHATWG, 2021.)

Edellisessä kappaleessa mainittu HTML-standardin kannanotto ”*Palvelimen on kuitenkin validoitava tiedot uudelleen, koska selaimen tietojen validointi voidaan helposti ohittaa.*” on merkittävä tämän tutkimusaiheen kannalta. HTML-standardissa ei oteta kantaa siihen, miten tämä palvelimen validointi tehdään.

2.3 HTML-syöttökenttien validointi

HTML-standardissa syöttölomakkeen kentät määritellään input -elementillä. Input -elementille on määriteltävissä attribuutteja, joilla määritellään syöttökentän tiedon arvon validointiominaisuuksia (WHATWG, 2021).

TAULUKKO 2 HTML INPUT -elementin attribuutit

Attribuutti	**Tyyppi**	**Kuvaus**
accept	Tiedosto	Tiedoston latauksessa ohjeellinen neuvo selaimelle. Kuvaa mitkä ovat sallitut tietotyypit tiedostojen latauksessa.
alt	Kuva	Kuvan tekstimuotoinen kuvaus tilanteisiin, joissa kuva ei ole näytettävissä.
autocomplete	Kaikki	Täydennetään syöttökenttään käyttäjän selaimessa säilytettyjä oletusarvoja, esimerkiksi osoitteita ja puhelinnumeroita. Arvoa käytetään myös piilotetuissa kentissä.
autocapitalize (globaali attribuutti)	Kaikki	Arvolla kontrolloidaan millä tavalla tekstisyöte muutetaan isoiksi tai pieniksi kirjaimiksi syöttövaiheessa. Oletusarvoisesti tekstin kirjaimet ovat pienellä kirjaimella. Seuraavat arvot ovat mahdollisia: • `off` tai `none`, ei muunnosta • `on` tai `sentences`, jokaisen lauseen ensimmäinen kirjain muutetaan isoksi kirjaimeksi ja loput ovat pienellä kirjaimella • `words`, jokaisen sanan ensimmäinen kirjain isolla ja loput pienillä kirjaimilla • `characters`, kaikki merkit kirjoitetaan isolla

checked	Radio Monivalinta	Monivalintalaatikoissa voidaan valita jokin syöttökenttä valmiiksi valituksi.
dirname	Teksti Hakukenttä	Attribuutissa kerrotaan sen kentän nimi, jossa välitetään tieto käyttäjän tekstinkirjoitussuunnasta (vasemmalta oikealle tai oikealta vasemmalle).
disabled	Kaikki	Kenttä on poistettu käytöstä niin, että siihen ei pysty syöttämään tietoa, muuttamaan arvoa, painamaan hiirellä tai valitsemaan kosketusnäytöllä.
form	Kaikki	Tällä attribuutilla voidaan kenttä liittää haluttuun lomakkeeseen esimerkiksi tilanteessa, jossa lomake sisältää alilomakkeita ja halutaan varmistaa, että kenttä liitetään haluttuun lomakekokonaisuuteen. Liittyy validointiin välillisesti.
formaction	Kuva Painonappi	Lomakkeen lähettämisen kohde, Uniform Resource Locator (URL).
formenctype	Kuva Painonappi	Lomakkeen merkistönkoodauksen tyyppi.
formmethod	Kuva Painonappi	HTTP-metodi, jota käytettiin lomakkeen lähettämisessä.
formnovalidate	Kuva Painonappi	Totuusarvo, jolla ilmoitetaan, ohitetaanko HTML-standardin mukainen lomakkeen kenttien validointi.
formtarget	Kuva Painonappi	Ilmoittaa mihin vastauksena saatu HTML palautetaan. Vaihtoehdot ovat: • _blank • _self • _parent • _top • *kehyksen nimi*
height	Kuva	Kuvan korkeus.
hidden (globaali attribuutti)	Kaikki	Totuusarvo, jolla ilmaistaan piilotettavat kentät. Selain ei näytä piilotettavaksi merkittyjä kenttiä.

list	Melkein kaikki	Arvo viittaa datalist-elementtiin, joka sisältää arvolistan, josta käyttäjä valitsee kenttään arvon.
		Esimerkki:
		`<input type="text" list="function-types">`
		`<datalist id="function-types">`
		`  <option value="function">function</option>`
		`  <option value="async function">async function</option>`
		`  <option value="function*">generator function</option>`
		`  <option value="=>">arrow function</option>`
		`  <option value="async =>">async arrow function</option>`
		`  <option value="async function*">async generator function</option>`
		`</datalist>`

max	Numeeriset tyypit	Kentän suurin arvo. Arvo voi olla numeerinen, päivämäärä tai kellonaika. `<input name=bday type=date max="1979-12-31">`
maxlength	Salasana Hakukenttä Puhelin Teksti URL	Syöttökentän maksimipituus.
min	Numeeriset tyypit	Kentän pienin arvo. Arvo voi olla numeerinen, päivämäärä tai kellonaika.

minlength	Salasana Hakukenttä Puhelin Teksti URL	Syöttökentän arvon minimipituus.
multiple	Sähköposti Tiedosto	Totuusarvo, joka kertoo, onko monen arvon valinta sallittu.
name	Kaikki	Kentän nimi. Nimi lähetetään palvelimelle lomakkeen lähetysvaiheessa.
pattern	Salasana Teksti Puhelin	Regexp-muotomääritys, jonka syöttöarvo pitää läpäistä, jotta tieto on oikea.
placeholder	Salasana Hakukenttä Puhelin Teksti URL	Attribuutti sisältää arvon, jota näytetään esimerkkiarvona tietoja syötettäessä. Tämän attribuutin arvoa ei käytetä oletusarvona, eikä arvolla ole merkitystä tiedon validoinnissa.
readonly	Melkein kaikki	Totuusarvo, jolla syöttökenttä voidaan asettaa lukumoodiin, niin että käyttäjä ei pääse tietoa muuttamaan.
required	Melkein kaikki	Totuusarvo, jolla määritellään syöttökenttä pakolliseksi.
size	Sähköposti Salasana Puhelin Teksti URL	Kentän koko.
src	Kuva	Resurssin URL-osoite.
spellcheck (globaali attribuutti)	Kaikki	Koodattu arvo, joka määrittelee pitääkö elementti tarkistaa kirjoitusvirheiden varalta. Seuraavat arvot ovat sallittuja: • `true`, ilmoittaa että elementti pitäisi mahdollisuuksien mukaan tarkistaa kirjoitusvirheiden varalta • `false`, ilmoittaa että elementtiä ei pitäisi tarkistaa kirjoitusvirheiden varalta
step	Numeeriset tyypit	Kenttäarvon portaittainen lisäysarvo
type	Kaikki	Kentän tietotyyppi: hidden, text, search, tel, url, email, password, date, month, week, time, datetime-local, number, range, color, checkbox, radio, file, submit, image, reset, button.
value	Kaikki	Kentän oletusarvo.
width	Kuva	Kuvan leveys.

Edellisessä taulukossa on syöttökenttien attribuutteja HTML-standardista ja yhdistettynä Mozilla-organisaation dokumentaatioon aiheesta kenttätyypitys (Mozilla Foundation, 2021; WHATWG, 2021).

Seuraavassa taulukossa syöttökentän tyyppi -attribuutissa ilmoitetaan kentän syöttötyyppi selaimen ja päätelaitteen näytön esitystapaa varten. Esimerkiksi HTML-lomakkeessa, jossa on syöttökentän tyyppi ”date” näytetään päätelaitekohtainen päivämäärän valintakontrolli. Tämä on tyypillisesti erilainen työasemaselaimessa kuin mobiilipuhelimessa. (Mozilla Foundation, 2021.)

TAULUKKO 3 Syöttökentän tyyppi -attribuutin avainsana, tietotyyppikuvaukset ja kontrollityyppikuvaukset

Syöttökentän tyyppi -attribuutti	**Tietotyyppi**	**Kontrollityyppi**
hidden	Mielivaltainen teksti, joka on piilotettu näytöltä. Hidden -tilassa olevan kentän tietoa ei validoida, mutta tieto lähetetään lomakkeen mukana palvelimelle.	n/a
text	Tekstikenttä, jossa ei ole rivinvaihtoja.	Tekstikenttä
search	Tekstikenttä, jossa ei ole rivinvaihtoja.	Hakukenttä
tel	Tekstikenttä, jossa ei ole rivinvaihtoja.	Tekstikenttä
url	Kokonainen URL.	Tekstikenttä
email	Sähköpostiosoite tai lista sähköpostiosoitteita.	Tekstikenttä
password	Tekstikenttä, jossa ei ole rivinvaihtoja. Arkaluontoista tekstiä. Esimerkiksi salasana, luottokortti, sosiaaliturvatunnus.	Tekstikenttä, joka peittää syötetyn tekstin
date	Päivämäärä (vuosi, kuukausi, päivä), ilman aikavyöhyketietoa.	Päivämäärän valinta
month	Päivämäärä, joka koostuu vuodesta ja kuukaudesta ilman aikavyöhyketietoa.	Kuukauden valinta
week	Päivämäärä, joka koostuu viikko ja vuosinumerosta ilman aikavyöhyketietoa.	Viikon valinta
time	Kellonaika (tunnit, minuutit, sekunnit ja sekunnin osat) ilman aikavyöhyketietoa.	Ajan valinta
datetime-local	Päivämäärä ja kellonaika (vuosi, kuukausi, päivä, tunnit, minuutit, sekunnit ja sekunnin osat), ilman aikavyöhyketietoa.	Päivämäärän ja ajan valinta
number	Numeerinen arvo.	Tekstikenttä tai valintakehä
range	Numeerinen arvo, jolla ilmaistaan likimääräinen tai arvorajat.	Liukuvalinta tai vastaava
color	sRGB-väriarvokoodi.	Värin valintaikkuna
checkbox	Arvolista, joka koostuu ennalta tarjotuista arvoista.	Valintaruutu
radio	Koodatut arvot.	Valintanappi

file	Nolla tai useampi tiedosto, joista ilmoitettu MIME-tyyppi ja vapaavalintaisesti myös tiedostonimi.	Kenttäotsikko ja painonappi.
submit	Kenttä, joka ilmoittaa viimeisen valitun koodatun arvon lomakkeen lähetysvaiheessa. Kentän painaminen aiheuttaa lomakkeen lähettämisen.	Painonappi
image	Koordinaatit (x,y), jotka lähetetään valitusta kuvasta palvelimelle lomakkeen lähetysvaiheessa. Kuvan painaminen käynnistää lomakkeen lähetyksen.	Joko painettava kuva tai painonappi
reset	Ei tietotyyppiä.	Painonappi
button	Ei tietotyyppiä.	Painonappi

2.3.1 Muunnosalgoritmit

HTML Living Standardin mukaan riippuen tietotyypeistä saattaa tietotyyppeihin olla liitettynä HTML-standardin mukaisia tietotyypin muunnosalgoritmeja. Arvon puhdistusalgoritmi (value sanitization algorithm) puhdistaa arvon alusta tai lopusta tyhjiä merkkejä tai rivinvaihtomerkkejä. Merkkijonosta numeroon -muunnosalgoritmi muuttaa tekstimuodossa olevan arvon numeeriseen muotoon. Numerosta merkkijonoon -algoritmi muuttaa numeerisen arvon merkkijonoksi. Merkkijonosta päivämääräolioksi -algoritmi muuttaa merkkijonomuodossa olevan päivämäärän päivämääräolioksi. Päivämääräolion muunnos merkkijonoksi -algoritmi muuttaa oliomuodossa olevan päivämäärän merkkijonoksi.

Muunnosalgoritmit laukeavat HTML-standardin mukaisesti kentän muutosvaiheessa automaattisesti. Näin esimerkiksi sähköpostin alkuun tai loppuun syötetyt tyhjät merkit poistetaan ja tiedon validointivaiheessa pelkkä sähköpostiosoite ilman tyhjiä merkkejä osoitteen alussa tai lopussa näytetään käyttäjälle ja viedään eteenpäin taustajärjestelmälle. (WHATWG, 2021.)

2.3.2 Säännöllisen lausekkeen tarkistukset

Input kenttään on mahdollista liittää säännöllinen lauseke (regular expression, RegExp), arvon tarkistamiseksi tai muotoilemiseksi. Säännöllinen lauseke pitää olla määritelty ECMAScript-standardin mukaisesti. Seuraavassa esimerkki syöttökentän säännöllisen lausekkeen tarkistussäännöstä. Syöttökenttään saa syöttää vain arvon, joka koostuu yhdestä numerosta, jota seuraa välittömästi kolme isoilla kirjaimilla kirjoitettua kirjainta:

```
<label> Osanumero:
 <input pattern="[0-9][A-Z]{3}" name="osanumero"
        title="Osanumero on yksi numero, jota seuraa kolme isolla kirjaimilla kirjoitet-
tua kirjainta."/>
</label>
```

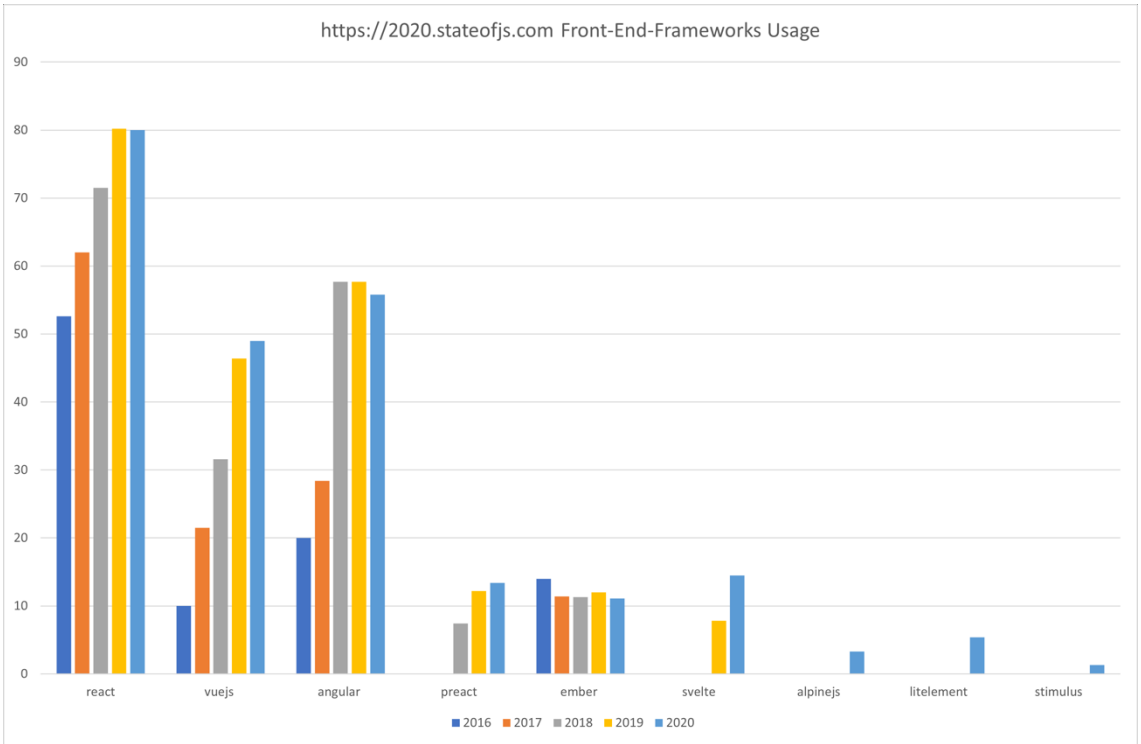
(WHATWG, 2021.)

2.4 Syöttötietojen validointi JavaScript-kehikoilla

Tässä luvussa kuvataan HTML-lomakkeen syöttötietojen validointi JavaScript-funktioilla. Edellisessä luvussa käytiin läpi HTML-lomakkeen syöttötietojen validointia HTML-standardin mukaisilla tavoilla. HTML-standardin mukaiset tavat eivät aina ole riittäviä tietojen rakenteelliseen tai monimutkaisempaan validointiin. Lomakkeiden syöttötietojen validointi JavaScriptillä on ollut Bajaj, Pattabiraman ja Mesbah (2014) mukaan yksi pääkeskusteluaiheista kehittäjien Stack Overflow -keskustelupalstoilla vuosina 2009—2012.

Yleinen trendi modernien web sovellusten kehityksessä on ollut siirtymä puhtaasta JavaScript-, HTML- ja CSS-pohjaisesta ohjelmistokehityksestä JavaScript Framework (JSF) ohjelmistokehikoiden käyttöön. Kuvaavaa kehityksen kannalta on huomata Gizasin, Christodouloun ja Papatheodoroun (2012) tutkimuksen olevan lähes irrelevantti kymmenen vuotta myöhemmin, koska käytettävien web sovellusten kehikoiden määrä ja suosio on muuttunut 10 vuodessa useaan otteeseen. Taivalsaaren, Mikkosen, Systän ja Pautassonin (2018) mukaan kehittäjien kehikoiden valinta on vaihtunut vuodesta toiseen muotitrendien omaisesti, sen mukaisesti mitä pidetään kunakin hetkenä kaikkein muodikkaimpana. Tällä hetkellä Vue.js, React ja Angular kehikot ovat kolme suosituinta web kehittäjien kehikkoa. (Taivalsaari ym., 2018.) Näistä Vue.js on puhdas JavaScript-kehikko, React määritykset käännetään JavaScript-koodiksi ja Angular perustuu TypeScript kieleen, joka käännetään Babel kääntäjällä JavaScript-koodiksi.

StateofJS-sivuston JavaScript-käyttäjäkyselyihin on viitattu useammassa tutkimuksessa. Sivusto ylläpitää kehittäjille suunnattua JavaScript-kehikoiden käyttöön, kiinnostukseen, tyytyväisyyteen ja käyttöaikomuksiin pohjautuvaa kyselyä ja niiden tuloksia. Tähän kandidaatintutkielmaan otettiin mukaan tuoreinta tietoa vuoden 2020 kyselytutkimuksesta verkkosivuston omasta analysointityövälineestä. Vuoden 2020 kyselyyn osallistui 23765 ihmistä 137 eri maasta. (StateofJS, 2021). Tämän vuoden tulosten perusteella nähdään kolme selkeästi eniten käytettyä JavaScript-käyttöliittymäkehikkoa (Front-End Framework): React, Vue.js ja Angular. Viime vuosina on erääksi uudeksi kiinnostuksen kohteeksi noussut Svelte, mutta se ei vielä näy kolmen eniten käytetyn joukossa. Tässä tutkimuksessa keskitytään kolmeen eniten käytettyyn JavaScript-kehikkoon: React, Angular ja Vue.js.



KUVIO 5 Vuoden 2020 käyttöliittymäkehikkojen käyttö (StateofJS, 2021)

2.4.1 React

React JavaScript -käyttöliittymäkehikko on Facebook-yhtiön (nykyisin Meta) kehittämä, Facebookin ja Instragramin web-sivustojen parempaa käyttäjäkokemusta parantaakseen. Facebook julkisti React:in avoimeksi koodiksi vuonna 2013. Tämän lisäksi Facebook julkisti vuonna 2015 React Native -kehikon tukeakseen mobiilisovelluksen kehitystä muun muassa iOS- ja Android-alustoilla. (Xing ym., 2019.)

React dokumentaation (2021) mukaan kehikossa on pääsääntöisesti kolme tapaa validoida syöttökentät lomakkeessa. Ensimmäinen tapa on nimeltään kontrolloimaton komponentti (uncontrolled component), jossa käytetään HTML DOM -validointia sellaisenaan. Tätä tapaa suositellaan käytettäväksi esimerkiksi siinä tapauksessa, että React-koodi pitää sovittaa ei-React-koodin kanssa ja tällöin yhteinen tekijä on selaimen oma DOM-puu. Toinen tapa on kontrolloitu komponentti (controlled component), jossa käytetään React:in sisäistä tilanhallintaa ja tapahtumakäsittelijöitä. Kolmantena tapana on käyttää ulkoisia laajennuksia, jotka on kehitetty nimenomaan lomakkeiden käsittelyyn ja validointiin. Esimerkkinä ulkoisesta laajennuksesta on Formik. Formik sisältää tuen muun muassa validationSchema-käsittelylle, jolloin validoinnissa voidaan käyttää omia validointikirjastoja tai esimerkiksi avoimen lähdekoodin Yup-validointiskeeman rakentajaa tietojen jäsentämiseen ja validointiin. (Formium, 2021; React, 2021.)

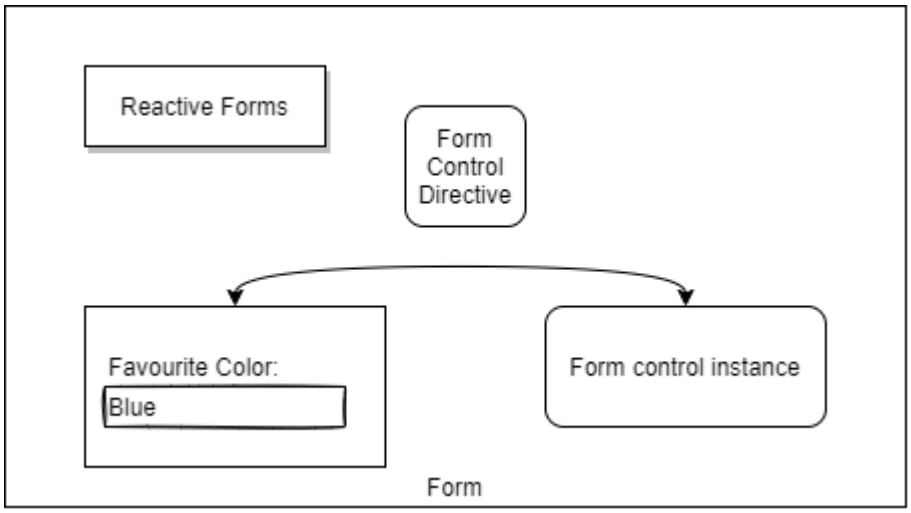
React:in lomakkeen rakentamiseen tarkoitetut kirjastot eivät sellaisenaan sovellu yleisen validointilogiikan käsittelyyn, koska lomakeolioon on sisään

rakennettu käyttöliittymän rakennusohjeet, ja viittauksia muihin lomakkeisiin on hankala joissakin tilanteissa tehdä.

2.4.2 Angular

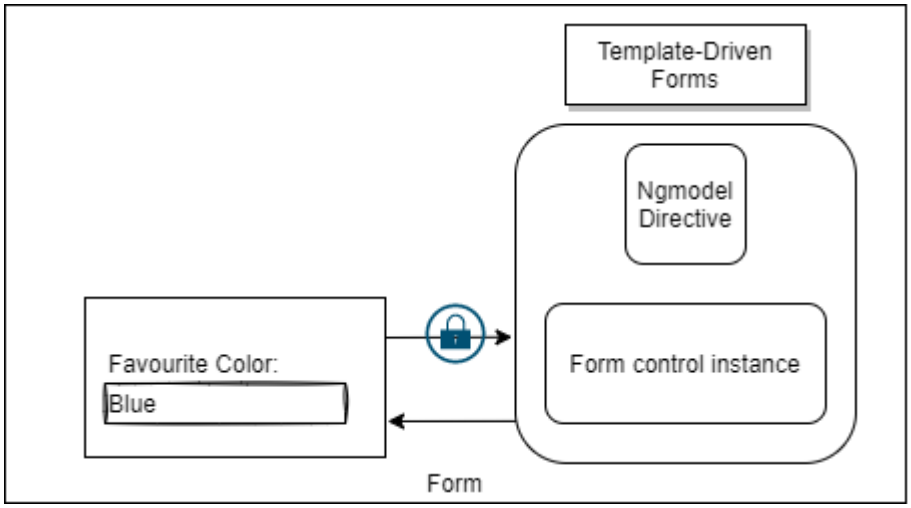
Xing, Huang ja Lai (2019) kuvaavat Angularin historiaa viittaamalla Ramos ym. (2018) tutkimukseen. Angular on Googlen vuonna 2010 kehittämä käyttöliittymäkehikko selainpohjaisten sovellusten kehittämiseen. Alkuperäinen Angular pohjautui JavaScript-kieleen, ja sisälsi lukuisia rajoitteita alkuperäisen suunnittelumallin puutteiden vuoksi. Vuonna 2016 Googlen kehitysryhmä uudelleen kirjoitti Angular2:in käyttämään TypeScript-kieltä. TypeScript-kieli on vahvasti tyypitetty kieli, joka käännetään erillisellä kääntäjällä JavaScript-koodiksi. Angular2 tukee myös paremmin mobiilikäyttöliittymiä pienemmän kokonsa ja nopeutensa vuoksi. (Xing ym., 2019.)

Googlen (2021) dokumentaation mukaan Angular tukee kahden tyyppisiä lomakkeita. Reaktiivisia lomakkeita (Reactive Forms) ja mallipohjaisia lomakkeita (Template-driven forms). Reaktiiviset lomakkeet tarjoavat suoran pääsyn lomakkeen oliomalliin. Verrattuna mallipohjaiseen lomakkeeseen, reaktiiviset lomakkeet ovat hyvin vakaita käyttää, skaalautuvampia, uudelleen käytettäviä ja testattavia. Niitä suositellaan käytettäväksi, jos syöttölomakkeet ovat tärkeä osa sovellustoiminnallisuutta.



KUVIO 6 Reaktiivinen lomake pitää lomakkeen oliomallin synkronisesti ajan tasalla

Mallipohjainen lomake perustuu HTML-lomakemalliin, jossa on Angular-direktiivejä ohjaamassa lomakkeen käyttöä. HTML-lomake luo muun muassa FormControl-instanssin implisiittisesti. Tätä FormControl-oliota ei pysty suoraan käsittelemään lomakkeesta. Mallipohjainen lomake soveltuu yksinkertaisiin lomaketarpeisiin, eikä se skaalaudu yhtä hyvin kuin reaktiivinen lomake.



KUVIO 7 Mallipohjainen lomake perustuu HTML-lomakemalliin

TAULUKKO 4 Reaktiivisen ja mallipohjaisen lomakkeen pääerot

	Reaktiivinen	Mallipohjainen
Lomakemallin määritys	Explisiittinen, luodaan komponenttiluokassa	Implisiittinen, luodaan direktiiveillä
Tietomalli	Rakenteellinen ja muuttumaton	Vapaarakenteinen ja muuttuva
Tiedon siirto näkymän ja mallin välillä	Synkroninen	Asynkroninen
Lomakkeen validointi	Funktioiden avulla	Direktiiveillä

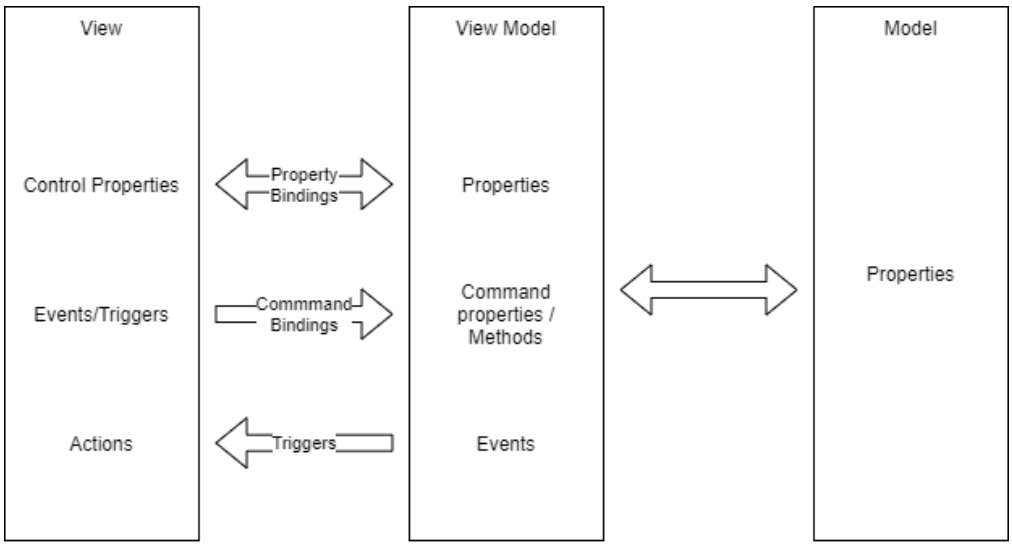
Angular-lomakkeen validointi tapahtuu lomakemallin tyypin mukaan. Reaktiivisessa lomakkeessa validointi tehdään suoraan lomakkeen oliomallissa, luokkaan liitettyjen funktioiden avulla. Angular kutsuu funktioita aina kun lomakkeen kentissä tapahtuu tiedon muutoksia. Mallipohjaisessa lomakkeessa tietojen validointi tehdään natiivin HTML-syöttökentän attribuuttien arvojen mukaisesti. Joka kerta kun kentän arvon lomakkeella muuttuu Angular kutsuu tietojen validointifunktioita Angular-direktiivien mukaisesti. Oletusarvoisesti Angular4+ poistaa HTML-lomakkeen validoinnin käytöstä lisäämällä novalidate -attribuutin syöttökentän lisäattribuutiksi. HTML-lomakkeen validointi saadaan käyttöön lisäämällä lisäattribuutti ngNativeValidate syöttökentän attribuuttiin. Angular-kehikossa on mahdollista luoda omia räätälöityjä validaatiofunktioita tietojen validointiin ja tietojen validointiin myös eri lomakkeiden välillä Angular-lomakeryhmän (FormGroup) avulla. (Google, 2021.)

Tämän tutkimuksen näkökulmasta pääero tietojen validoinnin kannalta on, että Angularin reaktiivinen lomake mahdollistaa paremmin JavaScript-pohjaisen validoinnin käytön. Mallipohjainen lomake tarkoittaisi, että iso osa validointilogiikkaa pitäisi koodata HTML-lomakemalliin, joka voi olla hankalasti uudelleenkäytettävissä esimerkiksi mikropalvelussa.

2.4.3 Vue.js

Vue.js on avoimen lähdekoodin progressiivinen JavaScript-ohjelmakehys SPA-käyttöliittymien rakentamiseen.Vue.js tukee kaikkia selaimia, jotka ovat ES5 (ECMAScript 2009) -yhteensopivia. Käytettäessä modernia moodia (Modern Mode) Vue.js sovellukset toimivat nopeammin moderneissa selaimissa, joissa käytetään ES2015+ JavaScript -versiota. Ensimmäinen versio 0.10 Vue.js kehikosta julkistettiin vuonna 2014. Kehikon kehitti Evan You ja hän on vieläkin aktiivisesti kehitystyössä mukana. Uusimmassa versiossa 3 on laajennettu TypeScript-tuki, tarjoten nyt myös virallisesti TypeScript-tuen. (Vuejs Github, 2013/2021.)

Vue.js perustuu niin kutsuttuun Model-View-ViewModel (MVVM) -malliin, jossa View-osuus on selaimen HTML-dokumentin DOM-rakenne, ViewModel-osuus on Vue.js kehikko ja Model-osuus on mikä tahansa JavaScript-olio (Xing ym., 2019).



KUVIO 8 Model-View-ViewModel (MVVM) -kehikko (Anderson, 2012)

Vue.js-kehikon lomakkeiden validointi perustuu joko puhtaaseen HTML-lomakkeen validointiin tai räätälöityyn validointiin. Räätälöity validointi puhtaasti Vue-kirjastolla perustuu muuttuneen tiedon validointiin JavaScript-koodissa tai asynkronisella kutsulla taustajärjestelmän REST/JSON-rajapintaan. Vue.js käyttöohjeissa viitataan myös kolmannen osapuolen kirjastoihin vuelidate ja VeeValidate, jotka tarjoavat tuotteistetut kirjastot lomakkeiden validointiin. (Vuejs.org, 2021.)

Vuelidate on ilmainen, yksinkertainen ja kevyt mallipohjainen validointikirjasto Vue.js kehikolle. Lomakkeen tietojen validointimalli määritellään sääntötiedostolla, jota Vuelidate hyödyntää tiedon validoinnissa. (Vuelidate, 2021.).VeeValidate on ilmainen lomakevalidointikirjasto, joka mahdollistaa lomakkeen tilan seurannan, synkroniset ja asynkroniset validoinnit, asynkroniset Ajax-kutsut, sekä synkroniset lomakkeen lähettämiset. VeeValidate voi myös

hyödyntää lomakkeen tietojen validoinnissa Yup- ja Zod-skeemavalidointia. VeeValidate tukee myös FormVueLate-lomakkeen generointiohjelmistoa. (VeeValidate, 2021.)

3 MIKROPALVELUJEN HTTP JSON - PALVELURAJAPINNAN TIETOJEN VALIDOINTI

3.1 Mikropalvelut

Dragonin, Giallorenzon, Lafuenten, Mazzaran, Montesin, Mustafinen ja Safinan (2017) mukaan mikropalveluarkkitehtuuri on viime vuosina tullut suureen suosioon järjestelmäarkkitehtuurisuunnittelussa. Mikropalveluarkkitehtuurissa järjestelmä on jaettu itsenäisiin moduuleihin, vastakohtana monoliittiselle arkkitehtuurille. Monoliittisessa järjestelmäarkkitehtuurissa järjestelmän moduulit jakavat samat resurssit (tietokoneen muisti, tietokanta, tiedostot), kun taas mikropalveluarkkitehtuurissa jokainen mikropalvelu on itsenäinen eikä suoraan riippuvainen toisen mikropalvelun resursseista. (Dragoni ym., 2017.)

3.1.1 Monoliittisten järjestelmien haasteet

Dragoni ym. (2017) luettelevat tutkimuksessaan useita syitä minkä takia monoliittisista järjestelmistä halutaan siirtyä mikropalveluarkkitehtuuriin: Isokokoiset monoliittiset järjestelmät ovat vaikeita ylläpitää ja kehittää niiden monimutkaisuuden takia. Virheenselvittely vaatii huolellista perehtymistä ohjelmakoodiin. Monoliittiset järjestelmät myös kärsivät moduulien välisistä suorista riippuvuussuhteista. Esimerkiksi kirjaston päivittäminen yhden moduulin muutoksen takia saattaa aiheuttaa käännösongelmia toisaalla. Muutokset jaetuissa resursseissa saattavat vaatia koko järjestelmän uudelleen käynnistystarpeen, aiheuttaen tarpeetonta alhaalla oloaikaa hidastaen kehitystä ja testausta.

Monoliittisten järjestelmien ajoympäristö ja teknologiat ovat yleensä osaoptimoituja johtuen siitä, että samaa järjestelmää käyttävät monet eri ristiriitaisia tarpeita omaavia moduuleja. Osa saattaa vaatia paljon muistia, osa paljon prosessointitehoa, jotkin moduulit vaativat relaatiotietokantaa ja toiset moduulit NoSQL-kantaa. Kehittäjät joutuvat tyytymään alustaan, joka on joko kallis tai

vain osaoptimoitu yksittäisiin tarpeisiin. Monoliittiset järjestelmät rajoittavat skaalautuvuutta. Tyypillisesti skaalautuminen tapahtuu saman sovelluksen kopioinnilla uudeksi instanssiksi ja kuorman jakamisella instanssien välillä kaikille toiminnallisuuksille, vaikka vain yksi moduuli olisi tarvinnut lisäkapasiteettia. Monoliittiset järjestelmät myös aiheuttavat teknologialukon kehittäjille. Kehittäjien täytyy käyttää samaa ohjelmointikieltä ja ohjelmointikehityksiä kuin muualla sovelluksessa.

Mikropalveluarkkitehtuurin on ehdotettu korjaavan monoliittisen arkkitehtuurin ongelmia. Dragoni ym. (2017) käyttävät mikropalveluita kuvatessaan termiä koossapysyvä, jolla he tarkoittavat sitä, että mikropalvelu toteuttaa palvelussaan toiminnallisuuden, joka on vahvasti sidonnainen vain siihen haasteeseen, johon palvelu on luotu. Määrityksen mukaan *mikropalvelu* on koossapysyvä, itsenäinen prosessi, joka keskustelee viestien välityksellä. *Mikropalveluarkkitehtuuri* on hajautettu sovellus, jossa kaikki moduulit ovat mikropalveluita. (Dragoni ym., 2017.)

3.1.2 Mikropalveluarkkitehtuurin vastaus monoliittisen järjestelmän haasteisiin

Dragoni ym. (2017) kuvaavat myös, miten mikropalveluarkkitehtuurilla ratkaistaan ongelmia, joita monoliittisessa järjestelmässä on todettu olevan. Mikropalvelut implementoivat rajoitetun määrän toiminnallisuutta, jonka vuoksi koodipohja on pienempi ja helpommin ylläpidettävissä ja korjattavissa. Koska mikropalvelut ovat itsenäisiä, voivat kehittäjät testata ja kehittää palvelua eristetysti vaikuttamatta muuhun järjestelmään. Uusia mikropalveluversioita voidaan tuoda käytössä olevan mikropalvelun rinnalle ja ottaa käyttöön vaiheittaisesti. Mikropalvelun muutos ei vaadi koko järjestelmän uudelleen käynnistämistä, koska uusi palveluversio voidaan asentaa vanhan rinnalle ja vanhaa palvelua käyttävät asiakasohjelmat voidaan muuttaa uuteen versioon vaiheistetusti.

Mikropalveluja ajetaan hyvin usein konttiteknologialla. Konttiteknologiat mahdollistavat ajoympäristön vapaan konfiguroinnin tarpeiden mukaisesti. Mikropalvelun skaalaaminen ei vaadi kaikkien mikropalveluiden ja niiden resurssien skaalaamista vaan yksittäistä mikropalvelua voidaan skaalata sen vaatiman tarpeen mukaan.

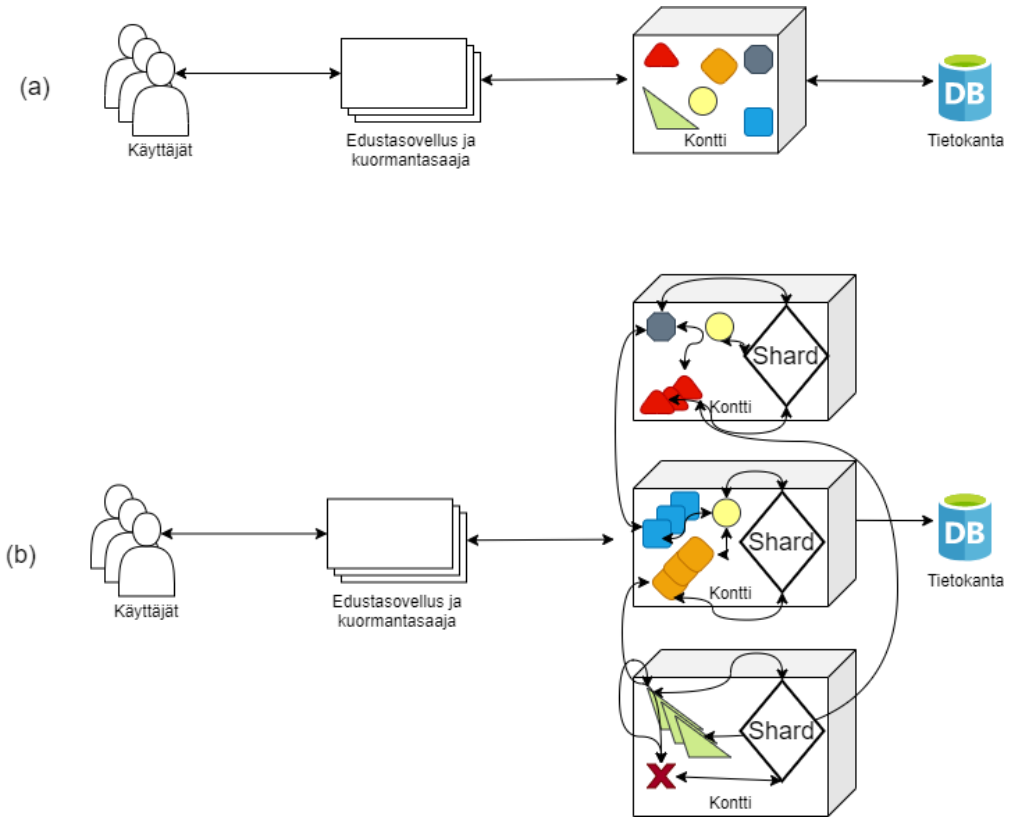
Ainut teknologialukko, joka mikropalveluarkkitehtuurista tulee, on teknologia, jota käytetään mikropalvelujen ja asiakasohjelmien välisissä kutsuyhteyksissä. Esimerkkeinä tästä tietoformaatti (esimerkiksi JSON), käytetyt protokollat (HTTP) ja datan merkistökoodaus (UTF8). (Dragoni ym., 2017.)

3.1.3 Mikropalveluiden tulevaisuuden suuntaukset ja haasteet

Jamshidi, Pahl, Mendoca, Lewis ja Tilkov (2018) kuvaavat mikropalvelun historiaa ja tulevaisuuden näkymiä tutkimuksessaan. Tutkimuksessa kuvataan yksityiskohtaisemmin mikropalveluarkkitehtuurin teknisiä suuntauksia lähtien

yksikertaisista konteissa ajettavista mikropalveluista, jatkuen hakupalvelun (Discovery Service) lisäämiseen palvelun edustalle, josta suuntaus on kohti sivuvaunu (sidecar) toteutuksia, joissa huolehditaan kutsuliikenteen hallinnasta ja vikasietoisuudesta. Uusin trendi on tuottaa logiikkapalvelut pilven FaaS (Function-as-a-Service) palveluina. (Jamshidi ym., 2018.)

Esposito, Castiglione ja Choo (2016) esittävät oikeutetusti kriittistäkin näkökulmaa mikropalveluarkkitehtuuriin. Mikropalveluiden määrän kasvaessa arkkitehtuurin monimutkaisuus lisääntyy erilaisten teknologioiden käyttöönoton myötä. Tietoturvamielessä hyökkäyspinta kasvaa moninkertaisesti verrattuna monoliittiseen arkkitehtuuriin. (Esposito ym., 2016.)



KUVIO 9 (a) Monoliittisen ja (b) mikropalvelupohjaisen arkkitehtuurin monimutkaisuus (Esposito ym., 2016)

Cerny, Donahoo ja Trnka (2018) tuovat esille mikropalveluarkkitehtuurin haasteet. Erityisesti tutkimuksessa nostetaan esiin useiden mikropalveluiden kehittäminen eri tiimeissä ja haasteet tietoformaattien, validointi- ja liiketoimintasääntöjen yhdenmukaistamiselle näissä tilanteissa. Ratkaisuksi esitetään tietämyksen ja säännöstön kuvaamista koneluettavassa muodossa, säännöstön manuaalista kopiointia palvelusta toiseen sekä mallipohjaisen (Model-Driven Development, MDE) kehityksen hyödyntämistä eri kehitystiimien välillä mallitiedon jakamiseksi. Vaikka mallipohjainen lähestyminen toisikin hyötyjä mikropalveluarkkitehtuurissa, kehittäjät keskittyvät yleensä vain oman alueensa mikropalvelun kehittämiseen, eikä ongelman selvittelyyn tai mallintamiseen yleisemmällä

tasolla. Cerny ym. viittaavat myös Lemos, Daniel ja Benatallah:in (2015) Web Services työväline- ja tekniikkatutkimukseen, jossa todetaan hajautetun kehityksen johtaneen mallinnuksen, analysoinnin ja päättelyn hajautumiseen, johtaen lopulta päästä-päähän suunnittelun hajoamiseen. (Cerny ym., 2018.)

3.1.4 Mikropalveluarkkitehtuurin näkökulma tässä tutkielmassa

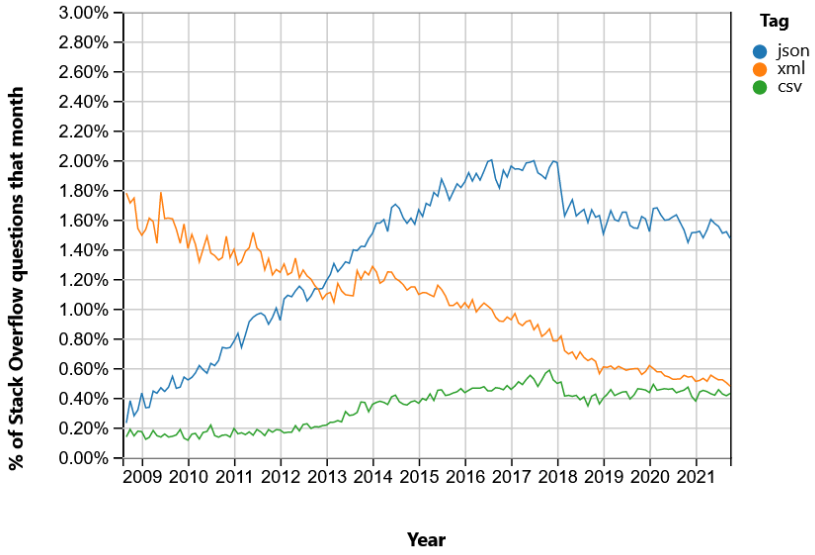
Viitaten edellisten lukujen perusteluihin on tähän tutkimukseen otettu lähtökohdaksi mikropalveluarkkitehtuuri, koska tällä hetkellä kasvava osuus järjestelmistä suunnitellaan tai rakennetaan mikropalveluarkkitehtuurin mukaisesti. Lisäksi oletetaan, että mikropalveluiden rajapinnat ovat rakennettu palvelukohtaisesti, johtaen siihen, että palvelurajapinnat eivät välttämättä ole keskenään yhdenmukaisia tietorakenteiden tai niiden validoinnin kannalta. Mikropalveluarkkitehtuuri mahdollistaa myös palvelukohtaiset muutokset, eli yhtenä päivänä toiminut rajapinta saattaa muuttua uuden version julkaisun myötä.

3.2 JSON yleiskatsaus

JSON (JavaScript Object Notation) on tietomuoto, joka perustuu JavaScript-ohjelmointikielen tietotyyppeihin. JSON-tietomuodon käytön lisääntyminen voidaan päätellä aihetta sivuavista tutkimuksista. Vargasin, Goelin, Steinerin ja Balasubramanian (2019) tutkimuksen mukaan Akamai-sisällönjakeluverkkoa (content delivery network, CND) käyttävien verkkosovellusten JSON-sisällön käyttö verrattuna HTML-sisällön käyttämiseen selainsovelluksissa oli moninkertaistunut pelkästään tutkimusjakson 04/2016—12/2017 aikana. Pääosa JSON-liikenteestä tuli mobiili- ja upotetuista laitteista. JSON-sisältöä käytettiin tutkimusjakson lopulla neljä kertaa enemmän kuin puhdasta HTML-sisältöä. (Vargas ym., 2019.) Rodríguezin, Baezin, Danielin, Casatin, Trabucconin, Canalin, Percannellan ym. (2016) tutkimuksessa analysoidaan Italian suurimman mobiili-internetoperaattorin Telecom Italian HTTP-palvelimen lokitietoja. Heidän tutkimuksensa pääkohde oli REST-rajapintojen käyttö. JSON-tietomuodon kannalta tutkimuksesta kävi ilmi, että vuonna 2016 hieman yli puolet REST-kutsuista oli JSON-formaatissa (51 %) verrattuna XML-formaattiin (49 %). Lokeista kävi selville myös JSON- ja XML-tietoformaattien erot: keskimääräinen JSON-tiedon koko oli 1545 tavua, kun taas XML oli 2606 tavua. JSON oli siis noin 40,7 % pienempi kooltaan. (Rodríguez ym., 2016.) Oumazizin, Belkhirin, Vacherin, Beaundryn, Blancin, Fallerin, Mohan ym. (2017) tutkimuksessa tutkittiin REST-rajapintojen käyttöä Android-mobiilisovelluksissa. Heidän kehittäjäkyselyssään 92.2 % suosivat JSON-formaattia muihin formaatteihin, XML:ään ja Comma Separated Value (CSV), verrattuna. (Oumaziz ym., 2017.)

Osana tätä kandidaatintutkielmaa analysoitiin JSON-formaatin suhdetta XML- tai CSV-formaatteihin käyttämällä Stack Overflow -palvelun Trends -analytiikkaa, johon syötteiksi annettiin nimikkeet ”json”, ”xml” ja ”csv”. Tämä

analytiikka tuottaa kaaviot vuodesta 2008 alkaen, näyttäen kuinka monessa kehittäjän kysymyksessä annettu nimike on ollut käytössä. (Stack Overflow, 2021.) Kaaviosta voitaisiin päätellä, että JSON on suhteessa XML- tai CSV-formaatteihin kehittäjien keskuudessa enemmän esillä.



KUVIO 10 Stack Overflow JSON-, XML- ja CSV-kehityskysymykset vuosilta 2009—2021

JSON-rakenteella on todettu olevan monia etuja verrattuna XML-formaattiin. JSON on nopeampi käsitellä ja vie vähemmän resursseja (Nurseitov ym., 2009). JSON myös kuluttaa vähemmän verkkoresursseja, koska tietoformaatin rakenteessa ei käytetä elementtinimiä XML:än tapaan, vaatien vähemmän tilaa saman tiedon kuljettamisessa (Lin ym., 2012). XML- ja JSON-rakenteilla on omat hyvät ja huonot puolensa. Dokumenteissa, joissa on paljon erilaisia tietotyyppejä ja elementtejä, XML on ideaalinen formaatti. JSON on paremmin sopiva dynaamisiin web-sovelluksiin (Ajax-pohjaisiin) ja yksinkertaisiin tiedonsiirtoihin. JSON:ia suositellaan käytettäväksi tiedonsiirtoihin, joita tehdään palvelimen ja web-sovellusten välillä (Simec & Maglicic, 2014). XML- ja JSON-rakenteiden ohjelmallisilla jäsentimillä on tyypillisesti iso merkitys tietorakenteen käsittelyyn. Mobiililaitteilla, joilla on työasemaa rajallisempi prosessointikapasiteetti, on todettu JSON-rakenteen soveltuvan parhaiten tietorakenteen jäsentämiseen esimerkiksi Java-olioiksi. Yksi iso tekijä on myös jäsentimen olemassaolo erilaisissa ohjelmointikielissä. JSON-jäsennin tyypillisesti löytyy lähes kaikista ohjelmointikielistä (Rodrigues ym., 2011).

JSON on formaattina yksinkertainen ja kooltaan pienempi kuin XML. Tämä voidaan havainnollistaa yksinkertaisella esimerkillä:

XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Opiskelijat>
  <Opiskelija>
```

```
    <Etunimi>Harri</Etunimi>
    <Sukunimi>Kaukovuo</Sukunimi>
    <Sahkoposti>harri.j.kaukovuo@student.jyu.fi</Sahkoposti>
  </Opiskelija>
</Opiskelijat>
```

JSON:

```
{
"Opiskelijat": {
  "Opiskelija": {
    "Etunimi": "Harri",
    "Sukunimi": "Kaukovuo",
    "Sahkoposti": "harri.j.kaukovuo@student.jyu.fi"
    }
  }
}
```

Edellä kuvatussa esimerkissä XML-formaatissa oleva tieto oli kooltaan 229 merkkiä ja JSON-formaatissa oleva tieto 160 merkkiä.

JSON on noussut myös suosituimmaksi tietorakenteeksi ohjelmallisille rajapinnoille (Application Programming Interface, API), yhdistettynä HTTP-protokollaan. JSON-formaattia käytetään paljon myös NoSQL- (esimerkiksi MongoDB) ja graph-tietokannoissa, jotka ovat nousseet suosioon vaihtoehdoksi relaatiokannoille. On käyttötapauksia, joissa JSON-formaatti ei vielä ole sopiva tiedon käsittelyyn samalla tavalla kuin XML. Esimerkiksi JSON-dokumentin kyselykieltä ei ole standardoitu, vaan käytössä on monia eri kyselykieliä. Viime aikoina on ollut kehitystä teoreettisemman pohjan rakentamiselle yleisen JSON-kyselykielen standardoimiseksi. (Bourhis ym., 2020.) Koska JSON on tällä hetkellä hallitseva tiedonsiirtoformaatti, on tämän tutkimuksen pohjaksi otettu olettama, että selaimen ja palvelimen liikenne on nimenomaan JSON-muodossa HTTP-protokollalla siirrettynä.

3.3 JSON-tietorakenteen validointi

JSON-tietorakenteen validointi on JSON-pohjaisen järjestelmäintegraation haaste. JSON-tietorakenteen validointi perustuu tyypillisesti skeemakieleen, jolla määritellään tietorakenteen oikeellisuussäännöt. JSON-teknologiassa on useampia vaihtoehtoja skeemakieliksi. Esimerkkejä skeemakielistä ovat JSON Schema, Walmart Labs Joi ja MongoDB Mongoose (Baazizi ym., 2019). Näistä lähimpänä virallista standardointia on JSON Schema.

Mikropalvelujen näkökulmasta JSON-tietorakenteen validointi voidaan tehdä kahdella tasolla: sovelluksessa ennen tietojen tallentamista tai tietokannassa osana tiedon lisäystä ja päivitystä. Tässä luvussa käydään ensin läpi skeemapohjaisia validointitapoja, esitellään ohjelmallinen tapa ja lopuksi käydään läpi tietokantojen JSON-validointitapoja. JSON-tietorakennetta on myös mahdollista validoida välimallien kautta muuntamalla JSON toiseen formaattiin,

esimerkiksi OWL:ään tai JSON-LD:hen, mutta näiden vaihtoehtojen läpikäynti rajataan tässä tutkimuksessa pois (Cheong, 2019; Jawaid ym., 2015).

3.3.1 JSON Schema

JSON Schema on sanasto, jonka avulla JSON-dokumentteja voidaan validoida ja niiden rakenteita ja sääntöjä dokumentoida. JSON Schema -määritystä kehittää vapaaehtoisten projekti json-schema.org -sivustolla. JSON Schema -määrityksen versio on tällä hetkellä Draft 2020-12 ja projektin tarkoituksena on jossain vaiheessa saattaa kehitystyö jonkin standardointiorganisaation alle tai jatkaa kehitystä jonkin aiheeseen liittyvän säätiön alaisuudessa (JSON Schema, 2021). Virallista ISO standardointistatusta JSON Schema -määrityksillä ei tällä hetkellä ole, mutta skeemamääritys on Internet Engineering Task Force (IETF) Internet-Draft -tilassa. Virallisen standardoinnin puuttuminen näkyy selkeimmin siinä, miten JSON Schema -määrityksiä tulkitaan ohjelmallisesti, eri validointiohjelmien tuottaessa erilaisia tuloksia. Pezoan, Reutterin, Suarezin, Ugartten ja Vrgočin (2016) tutkimuksessa samalla JSON Schema -määrityksellä eri tarkastusohjelmat tuottivat toisistaan poikkeavia tuloksia, mikä ei ole JSON Schema -määrityksen tarkoitus. He toteavat, että JSON Schema -määrityksen virallinen standardointi olisi tärkeätä yksiselitteisen määrityksen aikaansaamiseksi ilman tulkintaeroja. Alla olevassa taulukossa havainnollistetaan sitä, miten tutkimuksessa saatiin erilaiset validointitulokset. Taulukossa K-arvo ilmoittaa, että validointiohjelmisto tulkitsee JSON-dokumentin validiksi ja E-arvo sitä, että ohjelmisto tulkitsee JSON-dokumentin vialliseksi (epävalidiksi). Testitapaukset on merkitty T-arvolla ja eri validointiohjelmistot V-arvolla. (Pezoa ym., 2016.)

TAULUKKO 5 Neljän JSON-dokumentin validointi JSON Schema -määritystä (T) vastaan käyttäen viittä eri validointiohjelmistoa (V) (Pezoa ym., 2016)

	V1	V2	V3	V4	V5
T1	E	K	K	E	K
T2	K	E	K	E	K
T3	E	K	E	E	E
T4	-	-	E	-	-

K	Validi
E	Epävalidi
-	Tukematon

Esimerkki JSON Schema -määrityksestä (Droettboom & others, 2015):

```
{
  "type": "object",
```

```
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "birthday": { "type": "string","format": "date" },
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type": "string" }
      }
    }
  }
}
```

Fruth, Baazizi, Colazzo, Ghelli, Sartiani, Scherzinger, Grosmann ja Ram (2020) ovat tutkineet JSON Scheman tarkistusta JSON Schema Containment (JSC) -työvälineiden näkökulmasta. JSC-työvälineillä tarkoitetaan JSON-dokumentin tarkistusta JSON Schemaa vasten. Fruth ym. (2020) päätyivät samaan johtopäätökseen, että validointityövälineet eivät ole vielä valmiita. Ongelmia on erityisesti rekursion ja negaation validoinnissa. Toisaalta jos SchemaStore (JSON Schema Store, 2021) -määrityksistä poistetaan ne skeemat, joissa ei käytetä rekursiota tai negaatiota, jäävät jäljelle pääsääntöisesti sovellusten välisessä integraatiossa ja tiedon tallennusmuodossa käytettävät skeemat. Rekursiota ja negaatiota käytetään metatietoja ja konfiguraatiotietoja määrittelevissä skeemoissa (Fruth ym., 2020). Vaikka tutkimus viittaa ongelmiin validoinnissa, ei näitä ongelmia näyttäisi olevan nimenomaan siinä käyttötapauksessa, johon tässä kandidaatintutkielmassa viitataan. JavaScript-pohjaisista skeeman validointityökaluista Ajv tukee JSON Schema -määritystä validoinnissa (Ajv JSON Schema Validator, 2021).

Verrattaessa JSON-formaattia ja validointitapoja XML:ään, ei pidä unohtaa XML Schematron ISO standardia, joka määrittelee XML-rakenteen sääntöpohjaisen validointimekanismin. XML Schematronin avulla on mahdollista tehdä huomattavasti monipuolisempia tietosisällön ja rakenteen validointeja XML-rakenteelle. Ali (2019) on tutkimuksessaan selvittänyt JSON Scheman puutteita ja tehnyt ensimmäisen prototyypin XML Schematronia vastaavasta JSON Schematron -pohjaisesta validoinnista. Tämä prototyyppi löytyy vapaana lähdekoodina GitHub:ista ja npm moduulina Node.js käytettäväksi (Ali, 2021, 2019; jsontron, 2021). JSON-tietorakenteen sääntöpohjaisesta, XML Schematronin tyyppisestä, validoinnista on toistaiseksi vähän tutkimustietoa löydettävissä.

3.3.2 Joi

Joi on aiemmin Walmart Labs ja nykyisin Sideways Inc -yrityksen tukema avoimen lähdekoodin projekti. Joi on JavaScript-kirjasto ja skeemankuvauskieli, jonka avulla validoidaan JSON-dokumentteja (Sideway Inc, 2021).

Esimerkki Joi-skeemasta (Sideway Inc, 2021):

```
const Joi = require('joi');

const schema = Joi.object({
    username: Joi.string()
        .alphanum()
        .min(3)
        .max(30)
        .required(),

    password: Joi.string()
        .pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')),

    repeat_password: Joi.ref('password'),

    access_token: [
        Joi.string(),
        Joi.number()
    ],

    birth_year: Joi.number()
        .integer()
        .min(1900)
        .max(2013),

    email: Joi.string()
        .email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] } })
})
    .with('username', 'birth_year')
    .xor('password', 'access_token')
    .with('password', 'repeat_password');


schema.validate({ username: 'abc', birth_year: 1994 });
// -> { value: { username: 'abc', birth_year: 1994 } }

schema.validate({});
// -> { value: {}, error: '"username" is required' }

// Also -

try {
    const value = await schema.validateAsync({ username: 'abc', birth_year: 1994 });
}
catch (err) { }
```

Joi -kirjaston käytöstä on tällä hetkellä toistaiseksi vähän tutkimustietoa. Baazizi, Colazzo, Ghelli ja Sartiani (2010) ovat maininneet Joi:n käytöstä skeemavalidoinnissa, mutta yksityiskohtaisempia tietoja kirjaston ominaisuuksista tai vertailua esimerkiksi JSON Schemaan ei ole esitelty.

3.3.3 Mongoose MongoDB JSON-skeeman validointi

MongoDB on avoimen lähdekoodin NoSQL-tietokanta, joka perustuu JSON- dokumenttien tallennukseen. Sisäisesti MongoDB käyttää binääriformaatissa olevaa JSON-rakennetta, jota kutsutaan BSON:ksi. JSON-dokumenttien validointi MongoDB:ssä tehtiin vielä vuoteen 2016 asti ainoastaan Mongoose ODM -kirjastolla, mutta versiosta 3.4 lähtien MongoDB tukee suoraan JSON Schema -validointia tietokannassa Mongoose ODM:n lisäksi.

Mongoose on oliotietomallinnuskirjasto (Object Data Modeling, ODM) MongoDB tietokannan JSON-tietorakenteen määrittelyä ja validointia varten, Node.js alustalla. Mongoose ei ole osa MongoDB tietokantaa, vaan tätä kirjastoa käytetään Node.js sovelluksessa JSON-dokumenttien käsittelyssä ja validoinnissa. Alla esimerkki Mongoose-skeemamäärittelystä, jolla määritellään tietotyypit ja mahdolliset validointivirheilmoitukset.

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12,
    required: [true, 'Why no eggs?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea', 'Water',]
  }
});
```

(Mongoose, 2021.)

MongoDB JSON -tietokannan kanssa käytettynä validointivaihtoehdot ovat siis käytännössä Node.js sovelluksessa käytettävä Mongoose Node.js kirjasto tai tietokannan sisään rakennettu JSON Schema -pohjainen validointi (MongoDB, 2021).

3.3.4 JSON Type Definition (JTD)

Uutena tulokkaana JSON-dokumenttien validoinnissa on JSON Type Definition (JTD), joka on hyväksytty viralliseksi IEETF standardiehdotukseksi RFC8927. Tästä ei löytynyt tutkimusmateriaalia, koska standardiehdotus on hiljattain julkaistu. Tällä hetkellä standardointitaso on Experimental (Carion, 2020). Skeeman validointityökaluista JavaScript-kieleen perustuva Ajv tukee JSON Type Definition -skeemamääritystä validoinnissa (Ajv JSON Schema Validator, 2021).

3.3.5 Ohjelmallinen validointi

Tyypillinen tapa validoida JSON-tietorakenteita on käyttää mikropalvelun rakennuksessa käytettyä ohjelmointikieltä ja sen tarjoamia JSON-jäsenninkirjastoja. Json.org sivustolta löytyy lista ohjelmointikielistä ja niiden JSON-jäsentimistä. Marraskuussa 2021 listalta löytyi 169 erilaista JSON-jäsennintä.

Harrandin, Durieuxin, Bromanin ja Baudryn (2021) tutkimus toimii esimerkkinä erilaisista jäsentimistä ja niiden eroista. Tässä tutkimuksessa käytiin läpi 20 eri Java-kielen JSON-kirjastoa. Jäsenninkirjaston tehtävänä on lukea JSON-tieto, validoida sen oikeellisuus ja konvertoida Java-olioiksi ohjelmallista käyttöä varten. Jäsenninkirjasto myös generoi JSON-dokumentin halutuista Java-olioista. Lopputuloksena tästä tutkimuksesta on, että jäsentimet toimivat

hyvin eri tavalla, tuottaen vaihtelevia validointituloksia. Jäsentimet kyllä löytävät validointivirheet, mutta jokainen hieman eri tavalla, mikään näistä ei kuitenkaan 100 % RFC 8259 standardin mukaisesti. Tutkijaryhmän suositus onkin, että kattavaa JSON-validointia varten kehittäjien pitäisi validoida JSON-tietorakenne useamman Java JSON-jäsenninkirjaston kautta. Teoreettisesti, jos validoitaisiin kaikkien 20 jäsenninkirjaston kautta, päästäisiin 99,3 % kattavuuteen virheellisten JSON-testitiedostojen validointituloksissa. (Harrand ym., 2021.)

Ohjelmallisessa validoinnissa JSON-jäsentimen tehtävä on JSON-tietorakenteen validointi, mutta varsinainen sisällön semanttinen validointi jää sitten täysin räätälöidyn ohjelmalogiikan varaan sen jälkeen, kun JSON-rakenne on muunnettu ohjelman ymmärtämään sisäiseen muotoon.

3.3.6 JSON validointi tietokannassa

Mikropalveluissa JSON-tieto voidaan validoida tarvittaessa myös tietokannassa osana tiedon tallennusta. Alla olevaan taulukkoon on kerätty relaatio- ja NoSQL-tietokantoja, jotka tukevat JSON-tietotyyppiä. JSON-validointi on jaettu neljään sarakkeeseen. JSON RFC 4627, 7159 ja 8259 perustuvat IETF JSON-määritykseen tietorakenteen sisällöstä. Skeemavalidointi-sarakkeeseen on merkitty, tukeeko tietokanta jotakin JSON Scheman mukaista määrittelyä. CouchDB tukee JSON-dokumenttien validointia JavaScript-funktioilla, mutta tätä ei merkitty skeemavalidoinnissa Kyllä arvolla.

Tiedot on tarkistettu toimittajien ohjekirjoista marraskuussa 2021 ja myöhemmät muutokset ovat mahdollisia tuotteiden kehittyessä.

TAULUKKO 6 JSON-validointi tietokannoissa

Tietokanta	Tietokanta-tyyppi	**JSON RFC 8259** Validointi	**JSON RFC 7159** Validointi	**JSON RFC 4627** Validointi	Skeemavalidointi
Amazon DynamoDB	NoSQL				Ei
Azure CosmosDB	NoSQL				Ei
Couchbase	NoSQL				Ei
CouchDB	NoSQL				Ei
DB2 for Unix/Windows ver 11.5	Relaatio				Ei
DB2 for z/OS ver 12	Relaatio				Ei
MariaDB	Relaatio		Kyllä *)		Ei
Microsoft SQLServer	Relaatio		Kyllä *)		Ei
MongoDB	NoSQL				JSON Schema (3.4 ->)
MySQL	Relaatio		Kyllä		JSON Schema (8.0.17->)

Oracle 21c	Relaatio	Kyllä		Kyllä	Ei
PostgreSQL	Relaatio		Kyllä		Ei
			*)ei mainintaa mikä RFC		

Tietokannat lueteltua lähdeviitteineen:

- Amazon DynamoDB (Amazon, 2021)
- Azure CosmosDB (Azure, 2021)
- Couchbase (Couchbase, 2021)
- Apache CouchDB (Apache Foundation, 2021)
- IBM DB2 for Unix/Windows (IBM DB2 for Unix/Windows, 2021)
- IBM DB2 for z/OS (IBM DB2 for z/OS, 2021)
- MariaDB (MariaDB, 2021)
- MySQL (MySQL JSON, 2021)
- MongoDB (MongoDB, 2021)
- Microsoft SQL Server (SQLServer, 2021)
- Oracle (Oracle JSON, 2021)
- PostgreSQL (PostgreSQL, 2021)

Tietojen koostamisen jälkeen voidaan todeta, että JSON-validointi ei ole vielä tietokannoissa laajasti tuettu. JSON:in rakenteellisen validoinnin tuki on useammassa tietokannassa, mutta JSON Schema -validointia tukevat otokseen valituista tietokannoista vain MondoDB ja MySQL.

4 TOTEUTUSMALLEJA YHDENMUKAISEEN TIEDON VALIDOINTIIN

Tässä luvussa kuvataan yhdenmukaista tiedon validointilogiikan toteutusmalleja. Yhdenmukaisen tiedon validoinnin määritelmä kuvataan ensimmäiseksi. Toteutusmallit kuvataan ryhmitellen toteutusmallit niiden pääominaisuuksien mukaan: keskitetty validointi, lokaali validointi ja itsesäätyvä validointi. Validointilogiikan toteutusmallien ryhmittely on tässä tutkielmassa kehitetty, koska löydetyistä tutkimuksista ei löytynyt soveltuvaa ryhmittelyä.

4.1 Yhdenmukaisen tiedon validoinnin määritelmä

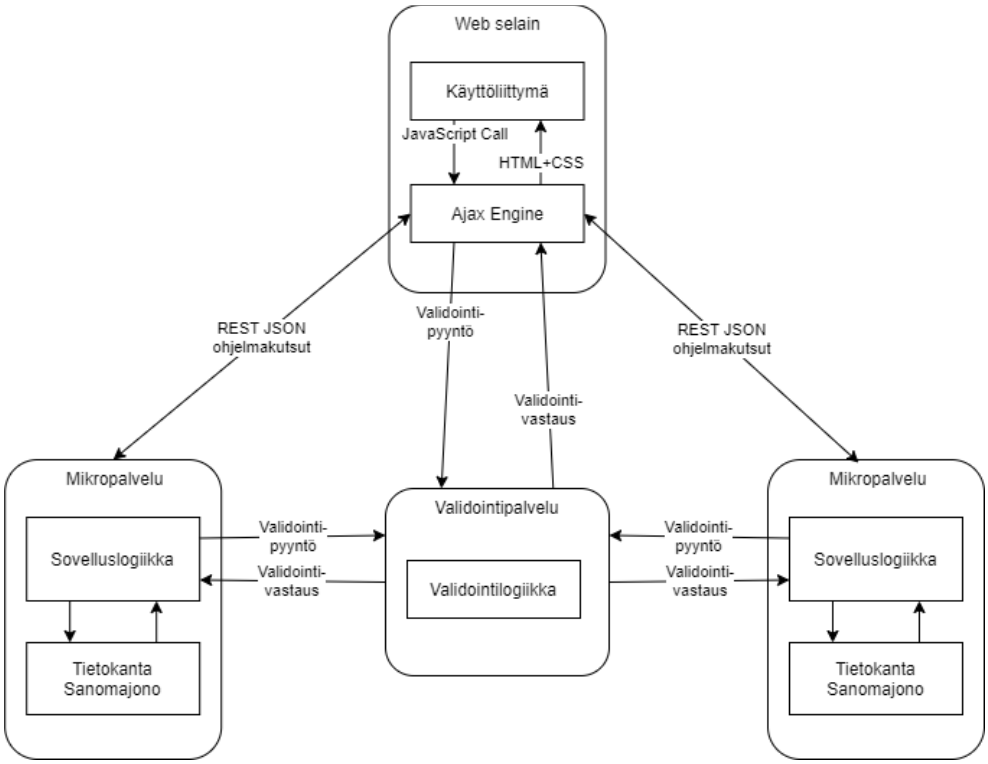
Tutkimusaiheena yhdenmukaisen validoinnin teemaa on tutkittu varsin vähän. Qi ja Guo (2015) ovat tutkineet asiaa tämän tutkimusongelman mukaisesti ja kutsuneet tutkimusongelmatiikkaa kaksinkertaiseksi validoinniksi (double validation). Qi ja Guon ehdotusta validoinnin yhdenmukaistamiseksi on kuvattu tarkemmin luvussa 4.3.1.

Fedorenko (2012) kuvaa asiakas-palvelin arkkitehtuurissa toteutettavaa asiakaspään syöttötietojen esivalidointia termillä DSL validointi ja validointisääntökone. Cheong (2019) esittelee OWL-aksioomeihin perustuvaa tiedon validointia järjestelmien välillä. Hän käyttää termiä yhdenmukainen tiedon validointi (unified data validation). Tutkimuksen määrittelyssä ei oteta kantaa siihen kuinka monta järjestelmää validointia tarvitsee. (Cheong, 2019.)

Tutkimusongelmana tällä kandidaatintutkielmalla on selvittää selainsovelluksen ja palvelimen yhdenmukaisen tiedon validointivaihtoehtoja. Vaikka Qi ja Guo (2015) käyttämä kaksinkertainen validointi voisi sopia tähän käyttötarkoitukseen, on geneerisempi termi tarpeen, koska kyse ei ole pelkästään ongelmatiikasta kahden järjestelmän välillä. Toisaalta taas DSL validointi tai sääntökonevalidointi ovat toteutustapoja, ei niitä voi käyttää yleisterminä validointiongelmatiikassa. Tässä tutkimuksessa päädyttiin käyttämään Cheongin termiä *yhdenmukainen tiedon validointi*.

4.2 Keskitetty validointi

Tässä luvussa kuvataan tapoja, joilla käyttöliittymän ja palvelinrajapinnan validointilogiikka voitaisiin pitää keskitettynä palveluna. Näin yksi validointilogiikkakoodi toteuttaisi kummankin tason validoinnin. Alla esitellään keskitetyn validoinnin looginen malli. Keskitetty validointi tässä tutkimuksessa tarkoittaa tapaa, jossa käyttöliittymä ja rajapinnat kutsuvat tiedon validointilogiikkaa keskitettynä palveluna. Alla olevassa mallissa ei oteta kantaa validointiteknologiaan. Validointipalvelu voi toteuttaa luvussa 1.1.5 esitettyjen validointitasojen 1—5 validoinnit. Tärkeätä on huomata, että mikropalvelut käyttäisivät tätä palvelua myös mikropalvelujen välisissä yhteyksissä. Tämä saattaa olla vastoin mikropalveluarkkitehtuurin perusperiaatteita, joihin yhtenä kuuluu riippumattomuus muista palveluista.



KUVIO 11 Keskitetyn validoinnin looginen malli

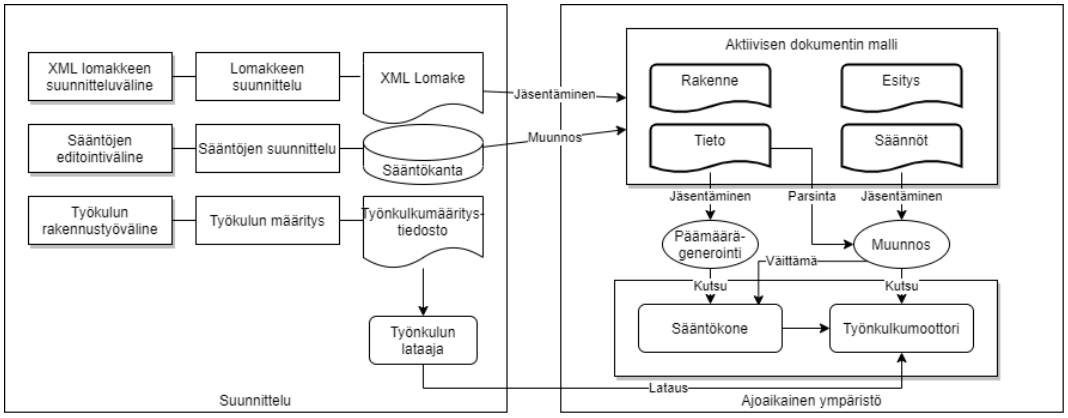
4.2.1 Sääntökone

Tässä luvussa käytetään yleistermiä *sääntökone* kuvaamaan toteutuksia, joissa validoinnin toteuttaa sääntöjen päättelymoottori. Sääntökoneella on myös monia englanninkielisiä nimiä. Puhutaan päättelykoneesta (reasoner tai inference engine), semanttisesta päättelykoneesta (semantic reasoner) tai ylipäätään sääntökoneesta (rule engine). Vertailua sääntökoneista on tehnyt muun muassa

Singh ja Karwayun (2010), jossa he vertailivat muun muassa Jess, Hoolet, Pellet, SHER, KAON2, RacerPro, Jena, FaCt, FaCt++, SweetRules, OWLIM, F-OWL ja BaseVISor sääntökoneita. Yksi yhteinen tekijä monelle näistä sääntökoneista oli tuki Semantic Web Rule Language (SWRL) sääntökielelle. (Singh & Karwayun, 2010.) SWRL-määritys jätettiin vuonna 2004 W3C organisaatiolle ehdotukseksi viralliseksi semanttisen verkon määrittelykielestä (Horrocks ym., 2004). SWRL-määritystä onkin pidetty sääntökoneiden yhteensopivuutta ajatellen tärkeänä askeleena (O'Connor ym., 2005). SWRL sisältää osan RuleML-määrittelykielestä. RuleML puolestaan on voittoa tuottamattoman organisaation RuleML Initiative:n ylläpitämä määrityskokoelma, jolla pyritään yhdistämään akateemisen ja yritysmaailman web-sääntömäärityksiä (Boley ym., 2010; RuleML Inc, 2021).

Ensimmäinen löydetty sääntökoneeseen pohjautuva lomakevalidoinnin tutkimus on Blandon (1999) ehdotelma teleoperaattoreiden tilaustensyötön tietojen validointi sääntökoneen avulla. Blando jaotteli tietojen validoinnin neljään tasoon. Lomaketiedon metatietojen tarkistus oli ensimmäinen taso, jossa tarkistettiin, että kaikki pakollinen ja ainoastaan lomakkeeseen liittyvä tieto on kutsussa mukana. Toinen taso oli syntaktinen kenttätason tarkistus. Kolmantena tasona oli semanttinen tarkistus, jossa tarkistettiin kenttien sisällön suhdetta toisiinsa. Neljäntenä tasona oli domain tason tarkistus, jossa tieto validoitiin järjestelmätasolla oikeaksi. Blandon toteutuksessa käytettiin itse kehiteltyä sääntökonetta omalla sääntökielellä, joskin he myös evaluoivat kolmannen osapuolen sääntökoneita, jotka perustuivat RETE-algoritmiin. (Blando, 1999.)

Nam, Jang ja Bae (2003) kehittivät aktiivisen dokumentin mallin, jossa dokumentin rakenne, käyttöliittymä, tieto ja tiedon käsittely eriytetään omiksi komponenteikseen. Tiedon käsittely oli toteutettu yhdistämällä käyttöliittymä, sääntökone ja työkulun järjestelmä. Tässä mallissa luotiin käyttöliittymä XML-määrityksestä muuntamalla lomakemääritys käyttöliittymäksi XSLT-muuntimen avulla. Lomakkeen logiikka ja validointisäännöt määritettiin Executable Rule Markup Language (ERML) avulla. ERML-määritys pohjautui XML-rakenteeseen ja ERML käännettiin Prolog-kielelle ja Prolog-sääntökoneella toteutettiin sääntöjen päättely. Tutkimuksessa vertailtiin ratkaisua aikaisempien vuosien aktiivisten dokumenttien tutkimuksiin, joita olivat tehneet useat tutkijaryhmät. Vertaillut toteutukset perustuivat 2000 luvun teknologioihin, joista selainpohjainen toteutus oli yksi vaihtoehto. Nam ym. esittivät oman ratkaisunsa olevan muita vaihtoehtoja parempi sen siirrettävyyden, nopean kehityksen ja muunneltavuuden vuoksi. (Nam ym., 2003.)



KUVIO 12 Aktiivisen dokumentin prosessoinnin rakenne (Nam Ym. 2003)

Terveydenhuoltoalalla on HL7 organisaatio ja sen ylläpitämät HL7 standardit olleet terveydenhuoltoalan de-facto standardeja. Jawaid, Latif, Mukhtar, Ahmad ja Raza (2015) esittelivät terveydenhuollon HL7 Fast Health Interoperability Resources (FHIR) standardin JSON-formaatin mukaisen datan validointia. FHIR JSON -formaatin validointiin ei ollut vuonna 2015 validointisäännöstöä. Jawaid ym. (2015) rakensivat Web Ontology Language (OWL) -sääntöihin perustuvan validointipalvelun, jossa JSON-dokumentti ensin konvertoitiin JSON for Linked Data (JSON-LD) -muotoon, josta sen rakenne validoitiin Pellet-sääntökoneen kirjastolla OWL-säännöstöä hyväksi käyttäen. (Jawaid ym., 2015). Jawaid ym. (2015) tutkimuksessa ei otettu kantaa mistä lähteestä JSON-dokumentti tälle sääntökoneelle tulee syötteenä. Se voisi siis olla joko käyttöliittymä tai toinen palvelu.

Yksi viimeisimpiä lähestymistapoja terveydenhuoltoalalla on perustunut HL7 järjestön uudempiin standardeihin. Prud'hommeaux, Collins, Booth, Peterson ja Solbrig (2021) esittelivät FHIR RDF -formaattiin perustuvan validointipalvelun, joka perustuu JSON-dokumenttien muuntamiseen Java-olioiksi, jotka puolestaan konvertoidaan ryhmän kehittämällä FHIT RDF -muuntimella RDF-muotoon. RDF-muodossa oleva tieto validoidaan W3C:en semanttisen verkon kehittämään Shape Expressions (ShEx) -säännöstöä vasten. (Prud'hommeaux ym., 2021). Myöskään Prud'hommeaux ym. eivät tutkimuksessaan ota kantaa siihen, mistä lähteestä JSON-dokumentti validointikoneelle saapuu.

Tutkimuksista voidaan päätellä sääntökoneen perustoiminnallisuudet pääpiirteissään kolmena päävaiheena: Sisään tulevan tiedon konvertointi sääntökoneen ymmärtämään muotoon. Tiedon validointi säännöstön pohjalta jakaen validointi kahteen osaan: rakenteellinen validointi ja sisällöllinen validointi. Palautetaan kutsuvalle ohjelmalle validoinnin tulos sellaisessa muodossa, että kutsuva ohjelma saa tiedon onnistumisesta tai virheellisistä tietomäärityksistä.

4.2.2 Keskitetty JSON Schema -validointi

Yksikertaisimmillaan keskitetty JSON-tiedon validointi perustuisi toteutustapaan, jossa selain lähettää lomakkeen tiedot JSON-tietomuodossa

validointipalveluun, validointipalvelu validoi JSON-dokumentin JSON Schema -määritystä vasten ja palauttaa vastauksena tiedon validoinnin tuloksesta.

Esimerkkinä eräästä toteutusmallista viittaamme Montesi ja Weber (2016) tutkimukseen mikropalvelujen tukipalveluista. Heidän mukaansa mikropalveluarkkitehtuurin käyttöönoton myötä esille on noussut useita käytännön ongelmia, joiden takia arkkitehtuuriin on kehitetty suunnittelumalleja, jotka auttavat ratkaisemaan näitä ongelmia: Katkaisijamalli (engl. circuit breaker), palveluhakemisto (engl. service discovery) ja rajapintayhdyskäytävä (API gateway). Katkaisijamallin tarkoituksena on estää mikropalvelun ylikuormittuminen kuormitustilanteessa. Palveluhakemiston tehtävä löytää ja ohjata asiakassovellus ottamaan yhteyttä oikeaan mikropalveluosoitteeseen. Rajapintayhdyskäytävän tarkoituksena tarjota keskitetty palvelu mikropalvelukutsuille, josta yhdyskäytävä toimii kutsun välittäjänä oikeaan palveluun. Usein rajapintayhdyskäytävä hoitaa myös katkaisijamallin ja palveluhakemiston tehtäviä. (Montesi & Weber, 2016).

Vaikka Montesi ym. (2016) eivät mainitse tiedon validointia rajapintayhdyskäytävän tehtävänä, olisi tämä luonnollinen paikka keskitetylle JSON-tietojen validoinnille. Tieteellisiä tutkimuksia rajapintayhdyskäytävän käytöstä JSON-tiedon validoinnissa ei löytynyt. Sen sijaan esimerkkejä kaupallisista toteutuksista on jo olemassa, esimerkiksi Amazon AWS API Gateway tukee JSON-tiedon validointia JSON Schemaa vasten (Amazon, 2021). Muita esimerkkejä ovat WSO2 Api Microgateway (WSO2, 2021) ja Oracle API Gateway (Oracle API Gateway, 2021).

Keskitetty JSON Schema -validointi toimisi rajapintayhdyskäytävässä niin, että koska sekä asiakassovellus, että myös muut mikropalvelut käyttävät samaa rajapintayhdyskäytävää, toimii keskitetty JSON-validointi tätä kautta. Haasteina saattavat tulla ongelmat JSON Scheman validointivirheiden raportoinnissa, eli kuinka kohdistetusti esimerkiksi käyttöliittymään välitetään virheilmoitus viallisesta tiedosta.

4.3 Lokaali validointi

Lokaalilla validoinnilla tarkoitetaan tässä tutkimuksessa tapaa, jossa validointi tehdään lokaalisti selaimessa ja lokaalisti mikropalvelussa. Käymme läpi kaksi vaihtoehtoista tapaa toteutukselle, joko käyttämällä samaa teknologiaa selaimessa ja mikropalvelussa tai käyttämällä eri teknologiaa selaimessa ja mikropalvelussa.

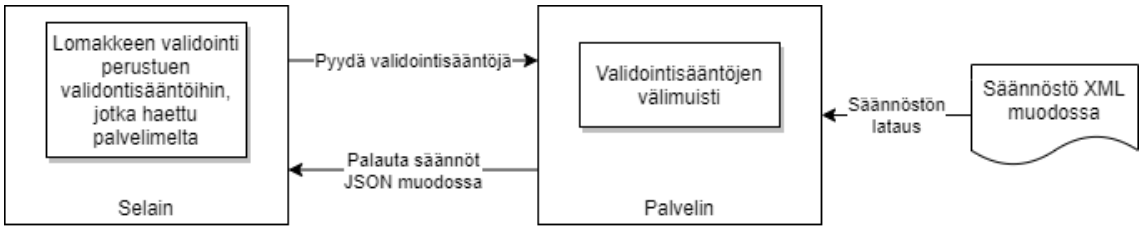
4.3.1 Samalla teknologialla toteutettu lokaali validointi

Tutkimuksen rajauksen mukaan keskitytään JavaScript-toteutuksiin selaimessa. Käytännössä tämä tarkoittaa JavaScript-teknologian käyttöä niin selaimessa kuin palvelimessakin. Käytännössä vaihtoehtoja on kaksi. Ensimmäisenä JSON -tietorakenteen validointi selaimessa ja mikropalvelussa, toisena saman JavaScript-koodin käyttö tietoalkioiden validoinnissa.

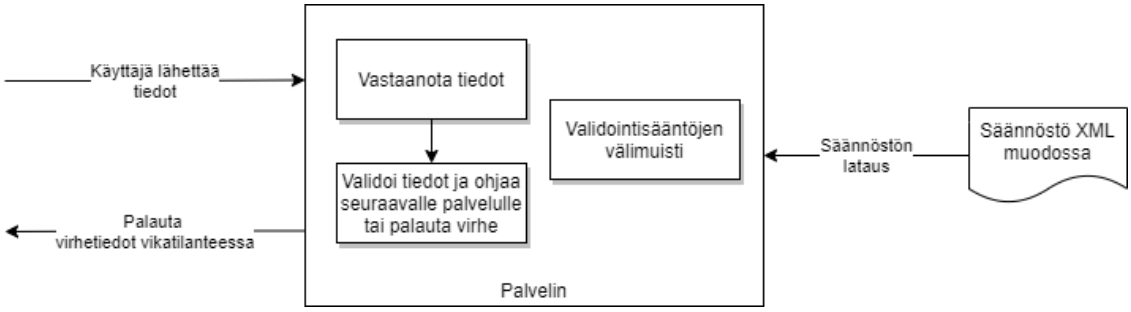
Kävin läpi JSON-tietorakenteen validoinnissa eri vaihtoehtoja ja ainut JavaScript-kirjasto, joka näyttäisi täyttävän vaatimuksen JavaScript-pohjaisuudesta sekä JSON Scheman validoinnista on Ajv-kirjasto. Sen avulla voidaan JSON-dokumentti validoida niin JSON Schema kuin JSON Type Definition -skeemamäärittelyjen mukaan niin selaimessa kuin palvelimellakin. Mikropalvelussa voidaan käyttää tekniikkana esimerkiksi Node.js. (Ajv JSON Schema Validator, 2021).

JavaScript-pohjaisista toteutuksista Dojo Toolkit täyttää tämän vaatimuksen ehdot. Dojo Toolkit (jatkossa Dojo) -kirjastolla voidaan toteuttaa validointikoodia, jota voidaan ajaa niin selaimessa kuin palvelimellakin. Palvelimessa tuettuna alustana on Node.js, jolla mikropalveluvalidointi pitäisi toteuttaa (Dojo Toolkit, 2021). Chen, Cao ja Mu (2011) ovat tutkineet Dojo-kirjaston käyttöä Ajax-pohjaisen dynaamisen SPA-sovelluksen rakentamisessa ja lomakkeen validoinnissa Dojo-kirjastolla. Dojo-kirjaston käytön kuvattiin yksinkertaistavan lomakkeen validointia, mutta Dojo-kirjaston kielioppia kuvattiin vaikeaksi (Chen ym., 2011). Kymmenessä vuodessa kirjasto on kehittynyt ja tätä pitäisi tutkia uudelleen, jotta selaimen että palvelimen uusimmat validointiominaisuudet saataisiin selvitettyä.

Qi ja Guo (2015) ovat tutkineet JavaScript-pohjaisten validointilogiikan käyttöä selaimessa ja palvelimella. Heidän ideansa on yhden koodin tekniikka, jossa samaa ohjelmakoodia ajetaan niin selaimessa kuin palvelimellakin, johtaen huomattavaan etuun koodin uudelleenkäytössä. Varsinaiset validointisäännöt olisivat XML-muodossa ja JavaScript-kirjasto tulkitsee sääntöjä selaimessa ja palvelimessa. (Qi & Guo, 2015). Pitää kuitenkin huomioida, että vaikka tutkimuksen ideat ovat hyviä, ei tutkimus ole saanut huomiota tiedeyhteisössä esimerkiksi viittausten määrässä mitattuna.



KUVIO 13 Selaimen validointi kaksinkertaisessa validoinnissa (Qi ym, 2015)



KUVIO 14 Palvelimen validointi kaksinkertaisessa validoinnissa (Qi ym, 2015)

4.3.2 Eri teknologialla toteutettu lokaali validointi

Skrupsky, Monshizadeh, Bisht, Hinrichs, Venkatakrishnan ja Zuck (2012) tutkivat mahdollisuutta selainpohjaisen validointilogiikan yhdenmukaistamiseen käyttämällä lähestymistapaa, jota he kutsuvat lyhenteellä WAVES (Web Application Validation Extraction and Synthesis). WAVES-tekniikassa kehittäjien täytyy ylläpitää validointikoodi ainoastaan palvelimen koodissa. WAVES-työvälineet käyvät palvelimen koodin läpi ja generoivat automaattisesti vastaavan tarvittavan JavaScript-koodin yhdenmukaista validointia varten. Tutkimuksessa WAVES-prototyyppi generoi palvelinsovelluksen PHP-validointikoodista JavaScript-koodiksi 83 %:sesti onnistuneesti. (Skrupsky ym., 2012.)

Chillón, Ruiz, Molina ja Morales (2019) kehittivät oman JSON skeeman louhintaan perustuvan Object-Document Mapper (ODM) koodigeneraattorin Mongoose- ja Morphia ODM -kirjastoille, todistaen että mallipohjaisilla työvälineillä on mahdollista generoida JSON-skeemamäärityksiä eri kielille: Mongoose JSON -skeemaksi sekä Morphia ODM Java -olioiksi. Samalla tutkimuksessa he mainitsevat kaupallisen Hackolade työvälineen, joka pystyy myös generoimaan graafisesta JSON-skeemamäärittelystä Mongoose ODM -mallin. Erona Hackoladen generointitapaan he mainitsevat, että Hackolade perustuu suunnittelumallipohjaiseen generointiin, kun heidän lähestymistapansa perustuu JSON-dokumenttien louhintaan ja sieltä generoituun malliin, joka ottaa kaikki JSON-dokumenttien variaatiot huomioon. (Chillón ym., 2019.)

Edellä mainittu työväline Hackolade mainitaan myös Fruth, Dauberschmidt, ja Scherzingerin (2021) tutkimuksessa, jossa esiteltiin Josch -työväline, jonka avulla voidaan kolmannen osapuolen liitännäisten avulla kutsua erilaisia JSON-skeeman louhintatyövälineitä, jotka palauttavat JSON Schema -määrityksen. Kaupallisen Hackoladen lisäksi yliopistomaailman Darwin mainitaan JSON-skeeman louhintavälineenä. (Fruth ym., 2021.)

Hacklolade on kaupallinen pioneeri NoSQL-kantojen ja REST-rajapintojen mallintamisessa. Aiemmin mainitussa Chillón ym. (2019) tutkimuksessa oli mainittu Hackloladen tukevan vain Mongoose ODM -mallin generointia. Tämä saattoi olla tilanne vuonna 2019, koska tällä hetkellä Hackolade tukee suunnittelumallin generointia esimerkiksi JSON Schema, YAML Schema, Avro Schema, Cassandra CQL, Ottoman Schema, MariaDB, Swagger tiedostoa ja yli 30 muuta määritystä. (Hackolade, 2021.)

Eri teknologialla toteutettu lokaali validointi tutkimusten perusteella perustuisi metamallinnukseen, jossa päämallista generoidaan eri ohjelmointikielille, ODM kirjastoille tai tietokannoille niiden käyttämät teknologiakomponentit tai määritystiedostot. Näin yhdestä mallista, jossa olisi määritelty myös validointisäännöt, voitaisiin generoida eri teknologioilla yhdenmukainen tietorakenne ja validointisäännöt.

4.4 Itsesäätyvä validointi

Itsesäätyvällä validoinnilla tarkoitetaan tässä kandidaatintutkielmassa visiota oppivasta järjestelmästä, joka oppisi käytetystä tiedosta tarvittavat validointisäännöt, vähintäänkin rakenteellisen validoinnin tasolla. Tähän liittyen yksi mielenkiintoinen lähestymistapa yhdenmukaiseen validointiin voisi olla hyödyntää automaattista JSON Schema louhintaa JSON-dokumenttien perusteella. Spoth, Kennedy, Lu, Hammerschemidt ja Liu (2021) esittelevät Apache Spark teknologiaan perustuvan data-analytiikka algoritmin JXPLAIN, jonka lopputuloksena massasta JSON-dokumentteja saataisiin muodostettua JSON Schema -määritys, joka kuvaa JSON-dokumenttien rakennetta ja validointisääntöjä (Spoth ym., 2021).

Samaan aihepiiriin kuuluu myös tutkimus, jossa Frozza, Mello ja Costa (2018) esittelevät oman mallinsa nimeltä JSON Schema Discovery ja tutkimuksessa myös vertailevat omaa menetelmäänsä paljon referoituun Wang ym. (2015) skeemagenerointiin. Frozza ym. (2018) olivat myös tehneet koosteen aiheen tutkimuksista alla olevan taulukon mukaisesti. (Frozza ym., 2018; Wang ym., 2015).

TAULUKKO 7 Skeeman generointitutkimuksen vertailu (Frozza ym., 2018)

	Klettke et al.	**Ruiz et al.**	**Wang et al.**	**Izquierdo et al.**	**Wischenbart et al.**	**Baazizi et al.**	**JSON Schema Discovery**
Tiedon lähde	MongoDB dataset	MongoDB, CouchDB, HBase	JSON datasets	Web service APIs	Social network APIs	JSON datasets	MongoDB dataset
Lähestymistapa	Hierarchical Summarization	MDE	Hierarchical Summarization	MDE	MDE	Hierarchical Summarization	Hierarchical Summarization and MDE
Syöttöformaatti	JSON	JSON	JSON	JSON	JSON	JSON	JSON and Extended JSON
Välimalli	SG	JSON Model			JSON Schema		RSUS
Tulosformaatti	JSON Schema	NoSQL DB Schema	eSiBu-Tree	ECORE Schema	ECORE Schema	Proprietary Schema	JSON Schema
Toteutustapa	Algorithm	Algorithm	Framework	Eclipse Plugin	Algorithm	Algorithm	SaaS

Näistä tutkimuksista Klettke ym. (2015) ja Frozza ym. (2018) lähestymistavat sopisivat adaptoituvan validoinnin käyttötapaukseen olettaen, että JSON Schema -validointi saataisiin integroitua saumattomasti käyttöliittymäkirjastoihin tietojen validoimiseksi.

De Lemos, Giese, Müller, Shaw, Andersson, Litoui, Schmerl ym. (2013) esittelevät vision itsesäätyvistä järjestelmistä, jossa järjestelmät oppisivat ajonaikaisesti verifioimaan ja validoimaan (V&V) toimintaansa. Ohjelmiston verifiointi ja validointi liittyy peruskysymykseen ohjelmiston laadusta sen elinkaaren ajan. Päätarkoituksena on varmistua siitä, että ohjelmisto täyttää ajonaikaisesti kaikki ne laatukriteerit, jotka sille on asetettu. (de Lemos ym., 2013.) Tiedon validoinnin näkökulmasta ohjelmiston pitäisi osata validoida tiedot aina ajantasaisesti. Tästä näkökulmasta skeeman louhinta voisi hyvinkin olla tulevaisuudessa teknologia, jota voidaan hyödyntää itsesäätyvien järjestelmien kehityksessä.

Koneoppimisen (Machine Learning, ML) näkökannalta Breck, Polyzotis, Roy, Whang ja Zinkevich (2019) esittelevät tiedon validointijärjestelmää koneoppimisen kielen sääntöjen korjaamiseksi. Koneoppimisen tekniikoita voitaisiin myös käyttää käyttöliittymän tietojen validointiin itsesäätyvästi. Biessmann, Golebiowski, Rukat, Lange ja Schmidt (2015) tutkivat koneoppimisen ongelmatiikkaa hieman laajemmasta näkökulmasta. Nykypäivänä organisaatiot turvautuvat enenemässä määrin koneoppimiseen ja tekoälyn käyttämiseen osana toimintaansa. Riippuen koneoppimisen algoritmeista, pienetkin ennakoimattomat muutokset sisään tulevassa tietosisällössä saattavat aiheuttaa ennakoimatonta käytöstä algoritmissa, jonka takia tulokseen ei voi luottaa tai tulos on epäoikeudenmukainen. Tämän takia tiedon validointi jokaisella tasolla olisi tärkeätä. Koneoppimisen komponenttien vastuullinen käyttö vaatii hyvin säädettyä ja skaalautuvaa tiedon validointijärjestelmää. Biessmann ym. (2015) jakavat validoinnin useampaan dimensioon, joista erityisesti oikeuden mukaisuuden validointi on ollut tekoälytutkimuksen suuren mielenkiinnon kohteena erityisesti tutkimuksessa mainitun Mehrabin, Morstatterin, Sazenan, Lerman ja Galstyan (2021) kokoavan tutkimuksen johdosta, jossa tutkittiin tekoalyjärjestelmien ihmisryhmiä syrjivien algoritmien vaikutuksia ja millä tavalla algoritmeja pitäisi kehittää oikeudenmukaisuuden saavuttamiseksi. Muita validointitasoja ovat skeeman validointi, tiedon oikeellisuus- ja johdonmukaisuusvalidointi, tilastollisten ominaisuuksien validointi, yksityisyyden validointi, opetetun mallin väärinkäytön validointi ja ihmisen suorittama manuaalinen validointi. (Biessmann ym., 2021).

TAULUKKO 8 Koneoppimisen validointitasojen määritelmä (Biessmann ym., 2021)

Validointitaso	**Kuvaus**
Skeeman validointi	Tiedon syntaktinen oikeellisuustarkistus (tietotyyppitarkistukset)
Tiedon oikeellisuus- ja johdonmukaisuusvalidointi	Tiedon validointi loogisten sääntöjen mukaisesti. Päätellään usein ohjelmallisesti tai sääntökoneiden avulla. Riippuen skeemoista voi skeemavalidointityökalu myös pystyä oikeellisuus- ja johdonmukaisuustarkistuksiin.
Tilastollisten ominaisuuksien validointi	Tietoa voidaan validoida tilastollisia ominaisuuksia hyödyntäen. Esimerkiksi: • virheellisen tiedon löytäminen tilastollisten avainlukujen perusteella • tilastollisten avaintietojen vertailu • poikkeavan tiedon löytäminen, poikkeava yhdistelmä tietoa
Oikeudenmukaisuuden validointi	Tieto validoidaan oikeudenmukaisuuden periaatteiden mukaisesti, ihmisryhmiä tai ihmistyyppejä syrjimättömästi.
Yksityisyyden validointi	Opetetuista malleista ei saa pystyä päättelemään millä ja kenen tiedolla mallia on opetettu.

Opetetun mallin väärinkäytön validointi (engl. robustness against adversarial attacks)	Mallin pitää pystyä päättelemään aiotaanko mallia väärinkäyttää opetuksessa käytettyjen tietojen avulla, päästäkseen haluttuun lopputulokseen.
Ihmisen suorittama manuaalinen validointi	Joissain tilanteissa on oltava mahdollisuus ihmisen manuaaliseen väliintuloon validoinnissa. Varsinkin tilanteissa, joissa mallit eivät pysty päättelemään oikeata lopputulosta.

Visiossa, jossa tieto validoidaan ensin käyttöliittymässä ja palvelinohjelmistossa on itsesäätyvä validointilogiikka, voitaisiin päästä tilanteeseen, jossa ohjelmallinen validointilogiikka on rakennettu pelkästään käyttöliittymään, ja johon taustajärjestelmä adaptoituu ajonaikaisesti, mahdollisesti tekoälyä hyväksi käyttäen. Esimerkkinä tästä voisi olla käyttötapaus, jossa tarpeeksi monta HTTP JSON -kutsua tietorakenteella, jossa on esimeriksi osoite aina syötetty, voisi palvelin adaptoitua vaatimaan osoitetietoa aina kaikilta kutsuilta.

5 YHTEENVETO

Tässä kandidaatintutkielmassa käytiin läpi yhtenäistä tiedon validoinnin haastetta nykyhetken muuttuvassa tietojärjestelmäympäristössä. Tutkielman tarkoituksena oli selvittää, kuinka validointimekanismeja voidaan hyödyntää käyttöliittymässä ja mikropalvelurajapinnoissa. Tämän seurauksena muodostui aihepiiriä kuvaava ja ilmentävä tutkimusongelma: Millä tavoilla ja teknologioilla voidaan rakentaa sovellus, jossa samaa validointilogiikkakoodia tai määritystä käytetään niin mikropalvelujen rajapintojen tiedon validoinnissa, kuin käyttöliittymässä?

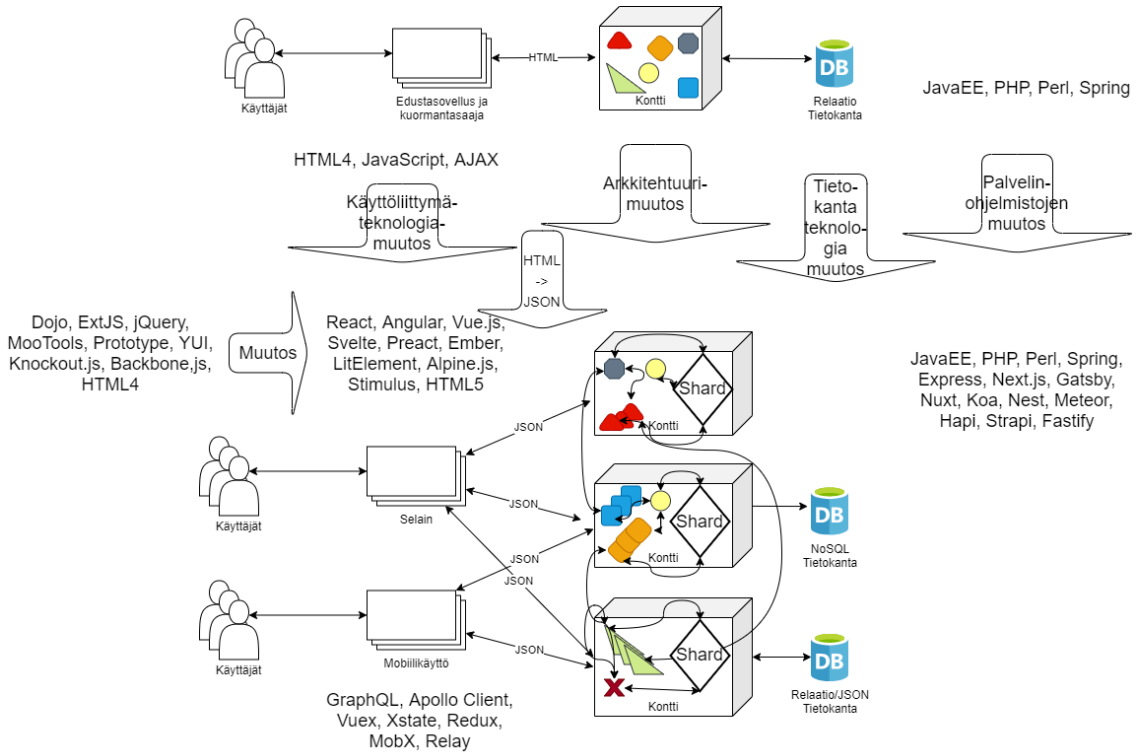
Aiheeseen liittyvää ongelmatiikkaa pohjustettiin kuvaamalla johdannossa käynnissä olevat tietojärjestelmäarkkitehtuurien ja teknologiasiirtymien haasteet. Haasteellisena nähdään lukuisten uusien käyttöliittymäteknologioiden esiinmarssi, ilman mainittavia standardointipanostuksia internet-teknologioiden edustusorganisaatioilta. Hyvä esimerkki tämän hetken jatkuvan muutoksen tilanteesta on HTML standardin muuttaminen kiinteästä versioidusta standardista jatkuvan muutoksen standardiksi, jota kutsutaan termillä ”HTML Living Standard”. Tutkimuksissa ja kaupallisissa materiaaleissa käytetään lyhennettä HTML5, jota ei oikeasti ole koskaan virallisesti standardoitu.

Käyttöliittymäteknologioiden muutossykli on nopea. Varsinkin uusien JavaScript tai TypeScript -pohjaisten kehikoiden lukumäärä kasvaa. Kehikoista syntyy uudelleenkirjoitettuja versioita. Tutkimusyhteisö tuntuu seuraavan toimettomana vierestä, koska muutokseen ei näytä voivan vaikuttaa, lisäksi standardointiorganisaatiot eivät pysy perässä. (Taivalsaari ym., 2018; Taivalsaari & Mikkonen, 2017a.)

Toisaalta palvelinarkkitehtuurissa on käynnissä siirtymä monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin. Yhteen sovitetusta monoliittisista palveluista muutos on kohti itsenäisiä mikropalveluita, joiden suunnitteluparadigmana on, että jokainen palvelu voi käyttää juuri niitä teknologioita, joita nähdään tarpeellisiksi, eikä muihin palveluihin pidä olla liikaa sidoksissa. Tästä mikropalveluarkkitehtuurin perusfilosofiasta syntyykin isoin looginen ongelma nimenomaan yhtenäisen validoinnin näkökulmasta. Mikropalveluita rakennetaan

lukuisilla erilaisilla teknologioilla ja niiden yhteen liittäminen, monitorointi, hallinta, tietoturvaaminen ja yhtenäinen suunnittelu on erittäin haasteellista. Vaikka mikropalveluarkkitehtuurissa onkin hyvät puolensa, on tiedeyhteisö vasta viime vuosina herännyt tuomaan esille myös huolenaiheita, joista tässä kandidaatin tutkielmassa mainitaan muun muassa Espositon, Castiglionen ja Choon (2016) tutkimus, jossa nostetaan monimutkaisuuden ongelmatiikka esille sekä suuren mikropalvelumäärän aiheuttama tietoturvariski laajan hyökkäyspinnan vuoksi. Cerny, Donahoo ja Trnka (2018) tuovat samoja monimutkaisuushaasteita esille tutkimuksessaan, jossa he myös esittävät saman tyyppisiä ratkaisuvaihtoehtoja kuin tässä kandidaatintutkielmassa, tosin pelkästään mikropalvelujen kontekstissa (Cerny ym., 2018). Zimmermann (2017) tuo esille tärkeän näkökulman mikropalveluihin: loppujen lopuksi mikropalvelut ovat vain yksi implementointikeino ja työtapa palvelukeskeisessä arkkitehtuurissa (Service Orientented Architecture, SOA). Monissa tutkimuksissa, kuten Salahin, Zemerlyn, Yeunin, Al-Qutayrin ja Hammadin (2016) mikropalvelun evoluutiotutkimuksessa annetaan ymmärtää, että mikropalvelut ovat monessa suhteessa SOA palveluita parempia ja seuraava taso hajautetussa arkkitehtuurissa (Salah ym., 2016). Näkemyksissä on selkeä ristiriita.

Alla olevassa kaaviossa on havainnollistettu käynnissä olevat muutokset, jotka vaikuttavat yhdenmukaiseen validointiin.



KUVIO 15 Käynnissä olevat muutokset, muunnelma (Esposito ym. 2016) mallista

Tutkimustyötä tehdessä kävi selväksi, että suoraan tutkimusaiheeseen liittyvää materiaalia löytyi vähän ja tutkimukset, jotka olivat aihepiiriltään sopivia, eivät olleet saaneet tutkijayhteisössä kovin laajaa huomiota. Tämä tarkoittanee joko sitä, että ongelmaa ei ole olemassa, tai että yhtenäisen validoinnin ongelma on liian monimutkainen tai teknologia liian nopeasti muuntuva tutkimuksissa käsiteltäväksi. Kuinka moni tutkija haluaisi panostaa sellaisten teknologioiden tutkimiseen, jotka saattavat olla epävalideja seuraavan kahden tai kolmen vuoden aikana?

Tulkitsevassa käsitetutkimuksessa lähdettiin liikkeelle käyttöliittymän validoinnista ja tavoista, joita nykyhetken selaimissa on käytettävissä. Selaimessa käytettävä HTML-kieli on luonnollinen lähtökohta, ja HTML:stä käytiin läpi sen standardivalidointiominaisuudet. Selaimen näyttöjen validoinnissa toinen tapa on käyttää JavaScript-kirjastoja. Käyttöliittymän JavaScript-kehikoista valittiin tällä hetkellä kolme eniten käytettyä: React, Angular ja Vue.js. Näissä näyttöjen validointia tehdään hieman eri tavoilla, mutta kaikkien vahvuudet ovat nimenomaan JavaScript-pohjaisessa validoinnissa. Tämän tutkielman perusteella käyttöliittymän validoinnissa standardinmukainen HTML5-näyttöjen validointi jää pieneen rooliin. Tämä saattaa olla merkki siitä, että HTML5 standardointi on jäänyt merkittävästi jälkeen käytännön tarpeista.

Mikropalvelujen validoinnissa aloitettiin kuvaamalla mikropalvelut ja arkkitehtuurillista muutosta monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin. Tutkimuksen aiheen rajauksena oli nimenomaisesti JSON-tietorakenteen käyttö käyttöliittymän ja rajapinnan välillä. JSON-rakennetta esiteltiin yleisesti XML-rakenteen korvaajana ja esiteltiin tutkimusmateriaaleihin pohjautuvat perusteet sen kasvaneelle suosiolle: pienempi koko, nopeampi käsittely ja natiivi tuki selaimissa. JSON-tietorakenteen validoinnista esiteltiin standardiehdotelmat JSON Schema ja JSON Type Definition. Lisäksi vaihtoehtoisina tapoina avoimen lähdekoodin skeemakirjastoja kuten Joi ja Mongoose. Ohjelmallinen validointi on varsin yleinen tapa, mutta ongelmana on JSON jäsenninkirjastojen vaihtelevat tulokset JSON-rakenteen tiedon validoinnissa. Lopuksi esiteltiin myös keinoja validoida JSON-tieto erilaisissa tietokannoissa. Tietokantojen JSON-tuesta ei löytynyt ajantasaista tutkimusaineistoa, joten tähän kandidaatin tutkielmaan käytiin läpi 12 erilaisen relaatio- ja NoSQL-tietokantojen JSON-validointituki. Lopputuloksena johtopäätös, että vaikka monessa tuotteessa oli tuki JSON-tiedon tallennukselle hakuominaisuuksilla, ainoastaan 41,6 %:ssa oli JSON rakenteellinen validointi ja 16,7 %:ssa JSON Scheman validointitoiminnallisuus.

Lopuksi tarkasteltiin toteutusmalleja yhdenmukaiseen tiedon validointiin. Viimeisen luvun sisältö vastaa tutkimuskysymykseen *"Millä tavoilla ja teknologioilla voidaan rakentaa sovellus, jossa samaa validointilogiikkakoodia tai määritystä käytetään niin mikropalvelujen rajapintojen tiedon validoinnissa, kuin käyttöliittymässä?"*. Yhdenmukaisen tiedon validoinnin termiä etsiessä työtä hankaloitti laaja-alaisen validoinnin tutkimuksen vähäisyys. Vaihtoehdoista päädyttiin käyttämään Cheong (2019) tutkimuksessa käyttämää termiä yhdenmukainen validointi (unified data validation). Itse Cheong tutkimus sinällään sisältää yhden toteutustavan tiedon validoinnin tasoista 1—2, jotka määriteltiin tämän kandidaatintutkimuksen johdannossa. Tähän kandidaatintutkielmaan rakennettiin oma

yhdenmukaisen tiedon validoinnin ryhmittely, koska kirjallisuuskatsauksista ei löytynyt vastaavaa ryhmittelykehikkoa. Ryhmittely jaettiin keskitettyyn validointiin, lokaaliin validointiin ja itsesäätyvään validointiin. Keskitetyssä validoinnissa käytiin läpi vaihtoehtoja, joissa validointi tehdään keskitettynä palveluna. Näitä olivat sääntökoneen käyttö ja keskitetty JSON Schema -validointi. Lokaalissa validoinnissa käytiin vaihtoehtoja läpi, jossa validointi suoritetaan käyttöliittymässä ja palvelimessa lokaalisti, mutta yhdenmukaisuutta noudattaen. Toteutusvaihtoehtoina saman teknologian käyttö ja eri teknologioiden käyttö. Itsesäätyvä validointi perustuisi täysin toisenlaiseen lähestymistapaan. Varsinaisesti tämän tyyppistä validointitapaa ei ole muissa tutkimuksissa esitelty, mutta idea johdateltiin ajatuksesta, jossa palvelin adaptoituisi käyttöliittymästä tuleviin tietovirtoihin itsesäätyvästi, esimerkiksi JSON Schema -louhinnan tekniikoilla tai tekoälymoottoreilla.

Tämän kandidaatintutkielman tutkimusaihe oli erittäin haastava sen laaja-alaisuuden ja käynnissä olevien muutosten takia. Eräs kirjallisuustutkimusta hankaloittava asia oli se, että monet tutkimuksista olivat vanhentuneita jo 5—10 vuoden jälkeen, eikä niitä voitu vanhentuneiden teknologioiden tai standardien vuoksi ottaa tutkimukseen mukaan. Aihealue itsessään on varsin tekninen, jonka vuoksi monia termejä, järjestelmäteknologian muutoksia ja jopa historiaa pidettiin tärkeänä käydä läpi osana taustaselvitystä ja aiheen pohjustusta.

Jatkotutkimusaiheina voitaisiin esimerkiksi selvitellä uuden mikroedusta-arkkitehtuurin (micro front-end) mahdollisuuksia tai uhkia esimerkiksi tiedon yhdenmukaisen käsittelyn suhteen (Micro-frontends, 2021). Toisena jatkotutkimusaiheen voisi olla empiirinen tutkimus siitä, millä tavoilla eri kehitysorganisaatiot huolehtivat yhdenmukaisesta validoinnista ja soveltuuko tässä kandidaatintutkielmassa esitelty ryhmittelymalli arviointikehikoksi tähän empiiriseen tutkimukseen. Kolmantena aiheena jatkotutkimukselle olisi kartoittaa uusien JavaScript-kehikoiden kanssa yleisesti käytetyt tilakoneet (state containers), näistä esimerkkinä React:in kanssa paljon käytetty Redux. Tilakoneilla voisi olla merkittäväkin rooli tiedon yhdenmukaisessa validoinnissa.

6 LÄHDELUETTELO

Ajv JSON Schema Validator. (2021). *Ajv JSON schema validator*. Haettu 16.11.2021 osoitteesta https://ajv.js.org/guide/environments.html

Ali, A. (2021, 13. marraskuuta). *Schematron based JSON Semantic Validator-GitHub*. Jsontron. Haettu 13.11.2021 osoitteesta https://amer-ali.github.io/jsontron/

Ali, A. (2019). Schematron-based Semantic Constraints Specification Framework and Validation Rules Engine for JSON. *2019 SoutheastCon*, 1–9. https://doi.org/10.1109/SoutheastCon42311.2019.9020470

Amazon. (2021, 13. marraskuuta). *Naming Rules and Data Types – Amazon DynamoDB*. Haettu 13.11.2021 osoitteesta https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.NamingRulesDataTypes.html

Amazon. (2021, 22. marraskuuta). *Enable request validation in API Gateway – Amazon API Gateway*. Haettu 22.11.2021 osoitteesta https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-request-validation.html

Anderson, C. (2012). The Model-View-ViewModel (MVVM) Design Pattern. Teoksessa C. Anderson (Toim.), *Pro Business Applications with Silverlight 5* (ss. 461–499). Apress. https://doi.org/10.1007/978-1-4302-3501-9_13

Apache Foundation. (2021, 13. marraskuuta). *Apache CouchDB Validation Function*. Haettu 13.11.2021 osoitteesta https://docs.couchdb.org/en/stable/intro/consistency.html?highlight=validation%20function#validation

Azure. (2021, 13. marraskuuta). *Working with JSON in Azure Cosmos DB*. Haettu 13.11.2021 osoitteesta https://docs.microsoft.com/en-us/azure/cosmos-db/sql/sql-query-working-with-json

Baazizi, M.-A., Colazzo, D., Ghelli, G., & Sartiani, C. (2019). Schemas and Types for JSON Data: From Theory to Practice. *Proceedings of the 2019 International Conference on Management of Data*, 2060–2063. https://doi.org/10.1145/3299869.3314032

Biessmann, F., Golebiowski, J., Rukat, T., Lange, D., & Schmidt, P. (2021). Automated data validation in machine learning systems. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*.

Blando, L. (1999). *A Framework for a Rule-Based Form Validation Engine*. http://www.se.rit.edu/~ateam/servlet@cmd=GetDiscourseFile&org.kelut.folium.sitemap.GroupBean.SELECTED_TAB=Forums&discourseFileId=65&filename=AFrameworkForARuleBasedFormValidationEngine-ISAS99.pdf

Boley, H., Paschke, A., & Shafiq, O. (2010). RuleML 1.0: The Overarching Specification of Web Rules. Teoksessa M. Dean, J. Hall, A. Rotolo, & S. Tabet (Toim.), *Semantic Web Rules* (ss. 162–178). Springer. https://doi.org/10.1007/978-3-642-16289-3_15

Bouras, C., & Konidaris, A. (2005). Estimating and eliminating redundant data transfers over the web: A fragment based approach. *International Journal of Communication Systems*, *18*(2), 119–142. https://doi.org/10.1002/dac.692

Bourhis, P., Reutter, J. L., & Vrgoč, D. (2020). JSON: Data model and query languages. *Information Systems*, *89*, 101478. https://doi.org/10.1016/j.is.2019.101478

Breje, A.-R., Gyorodi, R., Győrödi, C., Zmaranda, D., & Pecherle, G. (2018). Comparative study of data sending methods for XML and JSON models. *International Journal of Advanced Computer Science and Applications*, *9*. https://doi.org/10.14569/IJACSA.2018.091229

Busch, M., & Koch, N. (2009). Rich internet applications. *Stateof-the-Art. Rapp. tecn*, *902*, 52.

Carion, U. (2020). *JSON Type Definition* (Request for Comments RFC 8927). Internet Engineering Task Force. https://doi.org/10.17487/RFC8927

Casteleyn, S., Garrig'os, I., & Maz'on, J.-N. (2014). Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. *ACM Transactions on the Web*, *8*(3), 18:1-18:46. https://doi.org/10.1145/2626369

Cerny, T., Donahoo, M. J., & Trnka, M. (2018). Contextual understanding of microservice architecture: Current and future directions. *ACM SIGAPP Applied Computing Review*, *17*(4), 29–45. https://doi.org/10.1145/3183628.3183631

Chen, B., Cao, Y., & Mu, X. (2011). Data Validation Based on Dojo. *2011 International Conference of Information Technology, Computer Engineering and Management Sciences*, *4*, 126–129. https://doi.org/10.1109/ICM.2011.117

Cheong, H. (2019). Translating JSON Schema logics into OWL axioms for unified data validation on a digital manufacturing platform. *Procedia Manufacturing*, *28*, 183–188. https://doi.org/10.1016/j.promfg.2018.12.030

Chillón, A. H., Ruiz, D. S., Molina, J. G., & Morales, S. F. (2019). A Model-Driven Approach to Generate Schemas for Object-Document Mappers. *IEEE Access*, *7*, 59126–59142. https://doi.org/10.1109/ACCESS.2019.2915201

Couchbase. (2021, 13. marraskuuta). *Document | Couchbase Docs*. Haettu 13.11.2021 osoitteesta https://docs.couchbase.com/java-sdk/current/concept-docs/documents.html

de Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., … Wuttke, J. (2013). Software Engineering for Self-Adaptive

Systems: A Second Research Roadmap. Teoksessa R. de Lemos, H. Giese, H. A. Müller, & M. Shaw (Toim.), *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers* (ss. 1–32). Springer. https://doi.org/10.1007/978-3-642-35813-5_1

Di Zio, M., Fursova, N., Gelsema, T., Gießing, S., Guarnera, U., Petrauskienė, J., Quensel-von Kalben, L., Scanu, M., ten Bosch, K., van der Loo, M., & others. (2016). Methodology for data validation 1.0. *Essnet Validat Foundation, Brussels, Belgium*, 1–76.

Dojo Toolkit. (2021). *Dojo and Node.js — Dojo Toolkit Tutorial*. Haettu 16.11.2021 osoitteesta https://dojotoolkit.org/documentation/tutorials/1.10/node/index.html

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. Teoksessa M. Mazzara & B. Meyer (Toim.), *Present and Ulterior Software Engineering* (ss. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12

Droettboom, M. & others. (2015). *Understanding JSON Schema*. Haettu 12.11.2021 osoitteesta https://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf

ECMA International. (2017, 1. joulukuuta). *ECMA-404, The JSON data interchange syntax, 2nd edition, December 2017*. Ecma International. Haettu

25.9.2021 osoitteesta https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

Esposito, C., Castiglione, A., & Choo, K.-K. R. (2016). Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, *3*(5), 10–14. https://doi.org/10.1109/MCC.2016.105

ESS. (2018, 7. elokuuta). *ESS Handbook – Methodology for data validation version 2.0 – Revision 2018* [Text]. CROS - European Commission. Haettu 15.11.2021 osoitteesta https://ec.europa.eu/eurostat/cros/content/ess-handbook-methodology-data-validation-version-20-revision-2018_en

ESS. (2021, 15. marraskuuta). *European Statistical System (ESS) – European Statistical System (ESS) – Eurostat Overview*. Haettu 15.11.2021 osoitteesta https://ec.europa.eu/eurostat/web/european-statistical-system/overview

Formium. (2021, 10. marraskuuta). *Validation | Formik*. Haettu 10.11.2021 osoitteesta https://formik.org/docs/[...slug]

Fraternali, P., Rossi, G., & Sánchez-Figueroa, F. (2010). Rich Internet Applications. *IEEE Internet Computing*, *14*(3), 9–12. https://doi.org/10.1109/MIC.2010.76

Frozza, A. A., Mello, R. dos S., & Costa, F. de S. da. (2018). An Approach for Schema Extraction of JSON and Extended JSON Document Collections. *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, 356–363. https://doi.org/10.1109/IRI.2018.00060

Fruth, M., Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C., & Scherzinger, S. (2020). Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. Teoksessa G. Grossmann & S. Ram (Toim.), *Advances in Conceptual Modeling* (ss. 220–230). Springer International Publishing. https://doi.org/10.1007/978-3-030-65847-2_20

Fruth, M., Dauberschmidt, K., & Scherzinger, S. (2021). Josch: Managing Schemas for NoSQL Document Stores. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2693–2696. https://doi.org/10.1109/ICDE51399.2021.00306

Garrett, J. J. (2005). Ajax: A new approach to web applications, February 2005. *URL http://adaptivepath. com/ideas/essays/archives/000385. php*, *7*(3).

Google. (2021, 10. marraskuuta). *Angular – Validating form input*. Haettu 10.11.2021 osoitteesta https://angular.io/guide/form-validation

Hackolade. (2021, 22. marraskuuta). *Hackolade Forward-Engineering*. Haettu 22.11.2021 osoitteesta https://hackolade.com/help/ForwardEngineering.html

Harrand, N., Durieux, T., Broman, D., & Baudry, B. (2021). The Behavioral Diversity of Java JSON Libraries. *arXiv:2104.14323 [cs]*. http://arxiv.org/abs/2104.14323

Horrocks, Ian, Patel-Schneider, F, P., Boley, Harold, Tabet, S., Said, Grossof, Grosof, B., Dean, M., & Mike. (2004). SWRL: A Semantic Web rule language combining OWL and RuleML. *W3C Subm*, *21*.

IBM DB2 for Unix/Windows. (2021, 13. marraskuuta). *JSON application development for IBM® data servers*. Haettu 13.11.2021 osoitteesta https://www.ibm.com/docs/en/db2/11.5?topic=applications-json

IBM DB2 for z/OS. (2021, 14. huhtikuuta). *Using an SQL interface to handle JSON data in Db2 for z/OS*. Haettu 13.11.2021 osoitteesta https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/db2-for-zos/12?topic=dps-using-sql-interface-handle-json-data-in-db2-zos

Jadhav, M. A., Sawant, B. R., & Deshmukh, A. (2015). Single page application using angularjs. *International Journal of Computer Science and Information Technologies*, *6*(3), 2876–2879.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, *35*(3), 24–35. https://doi.org/10.1109/MS.2018.2141039

Jawaid, H., Latif, K., Mukhtar, H., Ahmad, F., & Raza, S. A. (2015). Healthcare Data Validation and Conformance Testing Approach Using Rule-Based Reasoning. Teoksessa X. Yin, K. Ho, D. Zeng, U. Aickelin, R. Zhou, & H. Wang (Toim.), *Health Information Science* (ss. 241–246). Springer International Publishing. https://doi.org/10.1007/978-3-319-19156-0_25

JSON Schema. (2021, 12. marraskuuta). *JSON Schema*. JSON Schema. Haettu 12.11.2021 osoitteesta http://json-schema.org/

JSON Schema Store. (2021, 12. marraskuuta). *JSON Schema Store*. Haettu 12.11.2021 osoitteesta http://schemastore.org

jsontron. (2021, 13. marraskuuta). *Jsontron NPM*. Npm. Haettu 13.11.2021 osoitteesta https://www.npmjs.com/package/jsontron

Karlberg, M. (2015, 9. tammikuuta). *ValiDat – Foundation* [Text]. CROS - European Commission. Haettu 15.11.2021 osoitteesta https://ec.europa.eu/eurostat/cros/content/validat-foundation_en

Lin, B., Chen, Y., Chen, X., & Yu, Y. (2012). Comparison between JSON and XML in Applications Based on AJAX. *2012 International Conference on Computer Science and Service System*, 1174–1177. https://doi.org/10.1109/CSSS.2012.297

MariaDB. (2021, 13. marraskuuta). *MariaDB JSON Data Type*. MariaDB KnowledgeBase. Haettu 13.11.2021 osoitteesta https://mariadb.com/kb/en/json-data-type/

Mesbah, A., & van Deursen, A. (2007). Migrating Multi-page Web Applications to Single-page AJAX Interfaces. *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 181–190. https://doi.org/10.1109/CSMR.2007.33

Micro-frontends. (2021, 21. marraskuuta). *Micro Frontends – Extending the microservice idea to frontend development*. Micro Frontends. Haettu 21.11.2021 osoitteesta https://micro-frontends.org/

MongoDB. (2021, 13. marraskuuta). *Schema Validation – MongoDB Manual*. Https://Github.Com/Mongodb/Docs-Bi-Connector/Blob/DOCSP-3279/Source/Index.Txt. Haettu 13.11.2021 osoitteesta https://docs.mongodb.com/manual/core/schema-validation/

Mongoose. (2021, 13. marraskuuta). *Mongoose ODM*. Haettu 13.11.2021 osoitteesta https://mongoosejs.com/

Montesi, F., & Weber, J. (2016). Circuit Breakers, Discovery, and API Gateways in Microservices. *arXiv:1609.05830 [cs]*. http://arxiv.org/abs/1609.05830

Mozilla Foundation. (2021, 13. marraskuuta). *Mozilla.org: The Input (Form Input) element – HTML: HyperText Markup Language | MDN*. Haettu 13.11.2021 osoitteesta https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input

MySQL JSON. (2021, 13. marraskuuta). *MySQL :: MySQL 8.0 Reference Manual: 11.5 The JSON Data Type*. Haettu 13.11.2021 osoitteesta https://dev.mysql.com/doc/refman/8.0/en/json.html

Nam, C.-K., Jang, G.-S., & Bae, J.-H. J. (2003). An XML-based active document for intelligent web applications. *Expert Systems with Applications*, *25*(2), 165–176. https://doi.org/10.1016/S0957-4174(03)00044-7

Nurseitov, N., Paulson, M., Reynolds, R., & Izurieta, C. (2009). Comparison of JSON and XML data interchange formats: A case study. *Caine*, *9*, 157–162.

O'Connor, M., Knublauch, H., Tu, S., Grosof, B., Dean, M., Grosso, W., & Musen, M. (2005). Supporting Rule System Interoperability on the Semantic Web with SWRL. Teoksessa Y. Gil, E. Motta, V. R. Benjamins, & M. A. Musen (Toim.), *The Semantic Web – ISWC 2005* (ss. 974–986). Springer. https://doi.org/10.1007/11574620_69

Oracle API Gateway. (2021, 22. marraskuuta). *Oracle API Gateway – JSON Schema Validation*. Haettu 22.11.2021 osoitteesta

https://docs.oracle.com/cd/E39820_01/doc.11121/gateway_docs/content/content_schema_json.html

Oracle JSON. (2021, 13. marraskuuta). *JSON in Oracle Database* [Topic]. Oracle Help Center; August2021. Haettu 13.11.2021 osoitteesta https://docs.oracle.com/en/database/oracle/oracle-database/21/adjsn/json-in-oracle-database.html#GUID-A8A58B49-13A5-4F42-8EA0-508951DAE0BB

Oumaziz, M. A., Belkhir, A., Vacher, T., Beaudry, E., Blanc, X., Falleri, J.-R., & Moha, N. (2017). Empirical Study on REST APIs Usage in Android Mobile Applications. Teoksessa M. Maximilien, A. Vallecillo, J. Wang, & M. Oriol (Toim.), *Service-Oriented Computing* (ss. 614–622). Springer International Publishing. https://doi.org/10.1007/978-3-319-69035-3_45

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., & Vrgoč, D. (2016). Foundations of JSON Schema. *Proceedings of the 25th International Conference on World Wide Web*, 263–273. https://doi.org/10.1145/2872427.2883029

Ping, Y., Kontogiannis, K., & Lau, T. C. (2003). Transforming legacy Web applications to the MVC architecture. *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, 133–142. https://doi.org/10.1109/STEP.2003.35

Pinto, C. M., & Coutinho, C. (2018). From Native to Cross-platform Hybrid Development. *2018 International Conference on Intelligent Systems (IS)*, 669–676. https://doi.org/10.1109/IS.2018.8710545

PostgreSQL. (2021, 11. marraskuuta). *PostgreSQL JSON Types*. PostgreSQL Documentation. Haettu 13.11.2021 osoitteesta https://www.postgresql.org/docs/14/datatype-json.html

Prud'hommeaux, E., Collins, J., Booth, D., Peterson, K. J., Solbrig, H. R., & Jiang, G. (2021). Development of a FHIR RDF data transformation and validation framework and its evaluation. *Journal of Biomedical Informatics*, *117*, 103755. https://doi.org/10.1016/j.jbi.2021.103755

Qi, M., & Guo, F. (2015). Research on Implementation Method of the Double Validation based on JavaScript. *2015 International Industrial Informatics and Computer Engineering Conference*, 641–645.

React. (2021, 10. marraskuuta). *Forms – React*. Haettu 10.11.2021 osoitteesta https://reactjs.org/docs/forms.html

Rodrigues, C., Afonso, J., & Tomé, P. (2011). Mobile Application Webservice Performance Analysis: Restful Services with JSON and XML. Teoksessa M. M. Cruz-Cunha, J. Varajão, P. Powell, & R. Martinho (Toim.), *ENTERprise Information Systems* (ss. 162–169). Springer. https://doi.org/10.1007/978-3-642-24355-4_17

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., & Percannella, G. (2016). REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. Teoksessa A. Bozzon, P. Cudre-Maroux, & C. Pautasso (Toim.), *Web Engineering* (ss. 21–39). Springer International Publishing. https://doi.org/10.1007/978-3-319-38791-8_2

RuleML Inc. (2021). *Introducing RuleML - RuleML Wiki*. Haettu 16.11.2021 osoitteesta http://wiki.ruleml.org/index.php/Introducing_RuleML

Salah, T., Jamal Zemerly, M., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 318–325. https://doi.org/10.1109/ICITST.2016.7856721

Sideway Inc. (2021, 12. marraskuuta). *JoiSite*. JoiSite. Haettu 12.11.2021 osoitteesta https://joi.dev/

Simec, A., & Maglicic, M. (2014). Comparison of JSON and XML Data Formats. *Central European Conference on Information and Intelligent Systems*, 272–275. http://www.proquest.com/docview/1629618564/abstract/49D49005FF884A3FPQ/1

Singh, S., & Karwayun, R. (2010). A Comparative Study of Inference Engines. *2010 Seventh International Conference on Information Technology: New Generations*, 53–57. https://doi.org/10.1109/ITNG.2010.198

Skrupsky, N., Monshizadeh, M., Bisht, P., Hinrichs, T., Venkatakrishnan, V. N., & Zuck, L. (2012). WAVES: Automatic Synthesis of Client-Side Validation Code for Web Applications. *2012 International Conference on Cyber Security*, 46–53. https://doi.org/10.1109/CyberSecurity.2012.13

Spoth, W., Kennedy, O., Lu, Y., Hammerschmidt, B., & Liu, Z. H. (2021). Reducing Ambiguity in Json Schema Discovery. *Proceedings of the 2021*

International Conference on Management of Data, 1732–1744. https://doi.org/10.1145/3448016.3452801

SQLServer. (2021, 13. marraskuuta). *Work with JSON data – SQL Server | Microsoft Docs*. Haettu 13.11.2021 osoitteesta https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver15

Stack Overflow. (2021, 21. marraskuuta). *Stack Overflow Trends*. Haettu 21.11.2021 osoitteesta https://insights.stackoverflow.com/trends?tags=json%2Cxml%2Ccsv

StateofJS. (2021, 20. marraskuuta). *State of JS 2020: Front-end Frameworks*. Haettu 20.11.2021 osoitteesta https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

Taivalsaari, A., & Mikkonen, T. (2017a). Return of the great spaghetti monster: Learnings from a twelve-year adventure in web software development. *International Conference on Web Information Systems and Technologies*, 21–44.

Taivalsaari, A., & Mikkonen, T. (2017b). The web as a software platform: Ten years later. *International Conference on Web Information Systems and Technologies*, *2*, 41–50.

Taivalsaari, A., & Mikkonen, T. (2011). The Web as an Application Platform: The Saga Continues. *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, 170–174. https://doi.org/10.1109/SEAA.2011.35

Taivalsaari, A., Mikkonen, T., Pautasso, C., & Systä, K. (2018). Client-Side Cornucopia: Comparing the Built-In Application Architecture Models in the Web Browser. *International Conference on Web Information Systems and Technologies*, 1–24.

Takala, T., & Lamsa, A. (2001). Tulkitseva kasitetutkimus organisaatio-ja johtamistutkimuksen tutkimusmetodologisena vaihtoehtona. *Liiketaloudellinen aikakauskirja*, 371–392.

Vargas, S., Goel, U., Steiner, M., & Balasubramanian, A. (2019). Characterizing JSON Traffic Patterns on a CDN. *Proceedings of the Internet Measurement Conference*, 195–201. https://doi.org/10.1145/3355369.3355594

VeeValidate. (2021, 10. marraskuuta). *VeeValidate*. Haettu 10.11.2021 osoitteesta https://vee-validate.logaretm.com/v4/

Vuejs Github. (2021). *Vuejs/vue* [JavaScript]. vuejs. Haettu 20.11.2021 osoitteesta https://github.com/vuejs/vue/blob/e78568ec20f7b34bbf655231df49b438c6279dbd/README.md (Original work published 2013)

Vuejs.org. (2021, 10. marraskuuta). *Form Validation — Vue.js*. Haettu 10.11.2021 osoitteesta https://vuejs.org/v2/cookbook/form-validation.html#top

Vuelidate. (2021, 10. marraskuuta). *Getting started | Vuelidate*. Haettu 10.11.2021 osoitteesta https://vuelidate-next.netlify.app/

Wang, L., Zhang, S., Shi, J., Jiao, L., Hassanzadeh, O., Zou, J., & Wangz, C. (2015). Schema management for document stores. *Proceedings of the VLDB Endowment*, *8*(9), 922–933. https://doi.org/10.14778/2777598.2777601

WHATWG. (2021). HTML Living Standard. *[Online] https://html.spec.whatwg.org/print.pdf*. Haettu 9.11.2021 osoitteesta https://html.spec.whatwg.org/print.pdf

WSO2. (2021, 22. marraskuuta). *API Schema Validation – API Microgateway 310 – WSO2 Documentation*. Haettu 22.11.2021 osoitteesta https://docs.wso2.com/display/MG310/API+Schema+Validation

Xing, Y., Huang, J., & Lai, Y. (2019). Research and Analysis of the Front-end Frameworks and Libraries in E-Business Development. *Proceedings of the 2019 11th International Conference on Computer and Automation Engineering*, 68–72. https://doi.org/10.1145/3313991.3314021

Yu, N., & Kong, J. (2016). User experience with web browsing on small screens: Experimental investigations of mobile-page interface design and homepage design for news websites. *Information Sciences*, *330*, 427–443. https://doi.org/10.1016/j.ins.2015.06.004

Zhao, Q., Liu, X., Chen, X., Huang, J., Teng, T., & Zhang, Y. (2010). Towards a data access framework for service-oriented rich clients. *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 1–8. https://doi.org/10.1109/SOCA.2010.5707150

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, *32*(3), 301–310. https://doi.org/10.1007/s00450-016-0337-0