Tampereen yliopisto

Laura Kurkinen

WEB-SOVELLUSTEN MODERNISOINTI YHDEN SIVUN SOVELLUKSIKSI ARKKITEHTUURINÄKÖKULMASTA

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastajat: Kari Systä
David Hästbacka
Kesäkuu 2022

TIIVISTELMÄ

Laura Kurkinen: Web-sovellusten modernisointi yhden sivun sovelluksiksi arkkitehtuurinäkökulmasta
Diplomityö
Tampereen yliopisto
Tietotekniikka
Kesäkuu 2022

Perinteiset web-sovellukset ovat rakenteeltaan monen sivun sovelluksia, joissa palvelin vastaa asiakkaan tekemiin palvelinpyytöihin kokonaisilla HTML-sivuilla. Tämän seurauksena koko käyttöliittymä on kerralla päivitettävä, jotta uusi sivu voidaan esittää käyttäjälle. Web-sovellusten sovellusalueiden laajentuessa ja toiminnallisuuden monimutkaistuessa monen sivun sovellusten toimintaperiaate alkoi kuitenkin aiheuttaa haasteita. Jatkuva yhteydenpito palvelimeen kokonaisten web-sivujen siirtämiseksi hidastaa sovelluksen toimintaa ja heikentää käyttökokemusta.

Ratkaisuna kohdattuihin haasteisiin ovat syntyneet yhden sivun sovellukset. Yhden sivun sovelluksissa palvelimen palauttama sisältö muodostuu esimerkiksi HTML-sivun osasta tai JSON-muotoisesta datasta kokonaisten HTML-sivujen sijaan. Asiakaskomponentissa suoritettava JavaScript-koodi vastaa vastaanotetun datan asettamisesta oikeaan kohtaan web-sivua. Tämä mahdollistaa sivujen osittaisen päivittämisen, jolloin koko käyttöliittymää ei ole tarpeen ladata uudelleen.

Nykyaikaisten web-sovellusten kasvaneet vaatimukset aiheuttavat tarpeen vanhentuneiden monen sivun sovellusten modernisoimiseksi. Modernisoinnilla viitataan kaikkeen sellaiseen toimintaan, jossa alkuperäinen sovellus pyritään muuntamaan kohdesovellukseksi muuttamatta merkittävästi sovelluksen toiminnallisuutta. Tässä diplomityössä tarkastellaan arkkitehtuurimuutoksia modernisointiprosessissa, jossa vanhentunut monen sivun sovellus muunnetaan yhden sivun sovellukseksi. Tarkoituksena on selvittää millaisia korkean tason muutoksia sovelluksen rakenteessa ja komponenttien vastuunjaossa ilmenee. Työ toteutettiin kirjallisuusselvityksenä verkkomateriaaleja, teknisiä dokumentaatioita, kirjoja ja tieteellisistä tietokannoista haettuja tutkimusartikkeleita hyödyntäen.

Modernisointimenetelmien ja muun soveltuvan aineiston tarkastelun pohjalta voidaan tunnistaa useita modernisointiprosessissa ilmeneviä arkkitehtuurimuutoksia. Merkittävimmät muutokset ovat asiakaskomponentissa suoritettavan sovelluslogiikan määrän kasvaminen, tapahtumapohjaisuuden korostuminen asiakkaan toiminnassa, sovelluksen tilanhallinnan monimutkaistuminen ja web-sivujen kokoamisen siirtyminen palvelimelta asiakkaalle.

Avainsanat: monen sivun sovellus, yhden sivun sovellus, modernisointi, ohjelmistoarkkitehtuuri, web-sovellus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

Laura Kurkinen: Renewal of legacy web apps to single-page apps: architecture viewpoint
Master of Science Thesis
Tampere University
Information Technology
June 2022

Traditional web applications are multi-page applications. In these applications, the server responds to requests with entire HTML pages. As a result, the entire user interface must be updated in order to show the new page to the user. However, as the application areas of web applications expanded and their functionality became more complex, this approach began to cause challenges. Constant communication with the server to transfer entire web pages slows down the application and impairs the user experience.

Single-page applications provide a solution to these challenges. In single-page applications, the content returned by the server consists of, for example, part of an HTML page or JSON data instead of entire HTML pages. JavaScript code executed by the client is responsible for placing the received data in the correct position on the web page. This allows the pages to be partially refreshed, eliminating the need to reload the entire user interface.

Increased expectations for modern web applications make it necessary to modernize legacy multi-page applications. Modernization refers to any activity that seeks to transform a source application into a target application without significantly altering the functionality of the application. This thesis examines the architectural changes in the modernization process in which a legacy multi-page application is converted into a single-page application. The thesis was carried out as a literature review using online materials, technical documentations, books and research articles from scientific databases.

Based on modernization methods and other appropriate materials, several architectural changes can be identified from the modernization process. The most significant changes are the increase in the amount of application logic executed by the client, the emphasis on event-based operations in the client, the complexity of state management and the transfer of the web pages rendering to the client.

Keywords: multi-page application, single-page-application, modernization, software architecture, web application

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

SISÄLLYSLUETTELO

1. Johdanto . . . 1
2. Peruskäsitteet . . . 3
   2.1 HTML JA CSS . . . 3
   2.2 URI . . . 4
   2.3 HTTP . . . 5
   2.4 JavaScript . . . 6
   2.5 JSON . . . 8
3. Web-sovellusten ohjelmistoarkkitehtuuri . . . 10
   3.1 Kerrosarkkitehtuurityyli . . . 12
   3.2 Asiakas-palvelin-arkkitehtuurityyli . . . 13
   3.3 REST . . . 14
   3.4 MV*-arkkitehtuurityylit . . . 17
4. Perinteiset ja modernit web-sovellukset . . . 19
   4.1 Monen sivun sovellukset . . . 19
   4.2 Yhden sivun sovellukset . . . 20
   4.3 RIA-sovellukset . . . 22
5. Web-teknologiat . . . 24
   5.1 JSP . . . 24
   5.2 React . . . 25
   5.3 Angular . . . 26
6. Modernisoinnin menetelmät . . . 29
   6.1 Yleiskatsaus . . . 29
   6.2 Menetelmien yksityiskohtaisempi tarkastelu . . . 32
      6.2.1 Käyttöliittymän mallipohjainen modernisointi . . . 33
      6.2.2 MIGRARIA-projekti . . . 34
      6.2.3 MVC-sovelluksen modernisointi . . . 36
7. Pohdinta . . . 40
   7.1 Tiedonhaku . . . 40
   7.2 Tulokset ja johtopäätökset . . . 41
8. Yhteenveto . . . 45

Lähteet . . . 47

1. JOHDANTO

Alkuaikoinaan internet muodostui joukosta toisiinsa linkitettyjä tekstidokumentteja. Jo 1990-luvun loppupuolella kuitenkin havaittiin, että yhdistämällä HTML-merkintäkieltä CSS-tyylien ja JavaScript-ohjelmointikielen kanssa voitiin toteuttaa aiempaa merkittävästi monimutkaisempaa toiminnallisuutta. [74] Web-selaimen ja -palvelimen välisen vuorovaikutuksen kehittyminen mahdollisti web-sivujen dynaamisen generoinnin palvelimella käyttäjän dataa hyödyntäen. Staattisista tekstidokumenteista siirryttiin varsinaisiin web-sovelluksiin, joissa selain toimii käyttöliittymänä palvelimella suoritettavalle ohjelmistolle. Tämän kaltaista perinteistä web-sovellusta, jossa palvelin vastaa kokonaisten HTML-sivujen kokoamisesta ja palauttamisesta selaimelle, kutsutaan monen sivun sovellukseksi. [34]

Web-sovellusten sovellusalueiden laajentuessa ja toiminnallisuuden monimutkaistuessa monen sivun sovellusten rajoitteet alkoivat kuitenkin aiheuttaa haasteita. Jatkuva yhteydenpito palvelimeen kokonaisten web-sivujen siirtämiseksi hidastaa sovelluksen toimintaa ja heikentää käyttökokemusta. [34] Myös sovelluksia kohtaan asetetut odotukset ovat merkittävästi kasvaneet. Nykyään web-sovellusten tulisi olla suorituskykyisiä, interaktiivisia ja käytettävissä useilla eri alustoilla, kuten mobiililaitteilla. Lisäksi sovellusten tulisi reagoida käyttäjän toimiin mahdollisimman nopeasti. [70]

Internet on kehittynyt yksinkertaisesta tekstidokumenttien jakamisen mahdollistavasta järjestelmästä vakavasti otettavaksi sovellusalustaksi, joka tarjoaa laajan valikoiman erilaisia palveluita ja ympäristön monipuolisen sisällön jakamiselle [74]. Taivalsaaren ja Mikkosen [74] vuoden 2011 ennusteen mukaan suurin osa loppukäyttäjien hyödyntämistä ohjelmistoista tullaan tulevaisuudessa kehittämään web-teknologioilla. Lisäksi monille käyttäjille web-selain saattaa olla jopa ainut tietokoneella käytettävä ohjelmisto [74].

Nykyaikaisten web-sovellusten kasvaneet vaatimukset aiheuttavat tarpeen vanhentuneiden monen sivun sovellusten modernisoimiseksi. Ratkaisuna kohdattuihin haasteisiin ovat syntyneet yhden sivun sovellukset, joissa kokonaisten HTML-sivujen sijaan palvelin vastaa selaimelle vain sivun osalla tai esimerkiksi JSON-muotoisella datalla [71]. Modernisoinnilla viitataan kaikkeen sellaiseen toimintaan, jossa alkuperäinen sovellus pyritään muuntamaan kohdesovellukseksi muuttamatta merkittävästi sovelluksen toiminnallisuutta. Uutta sovellusta ei rakenneta alusta asti uudelleen. Pohjaksi otetaan vähintään lähdesovelluksesta laadittu malli, johon muunnosprosessi kohdistetaan.

Tässä diplomityössä tarkastellaan arkkitehtuurimuutoksia modernisointiprosessissa, jossa vanhentunut monen sivun sovellus muunnetaan yhden sivun sovellukseksi. Tarkoituksena on selvittää, millaisia korkean tason muutoksia sovelluksen rakenteessa ja komponenttien vastuunjaossa ilmenee, sekä millaisilla menetelmillä muunnosprosessi voidaan suorittaa. Näin ollen useammasta lähteestä pyritään induktiivisen päättelyn avulla muodostamaan yhteisiä johtopäätöksiä. Hyvin harvoissa lähdemateriaaleissa esitetään numeerisia tuloksia menetelmien toimivuudesta, joten modernisointimenetelmiä on analysoitu laadullisesti. Työ on toteutettu kirjallisuusselvityksenä verkkomateriaaleja, teknisiä dokumentaatioita, kirjoja ja tieteellisistä tietokannoista haettuja tutkimusartikkeleita hyödyntäen.

Luvussa 2 esitellään lyhyesti web-sovellusten toiminnan kannalta olennaisia peruskäsitteitä. Luvussa 3 perehdytään ohjelmistoarkkitehtuurin käsitteeseen ja tutustutaan web-sovelluksissa käytettäviin arkkitehtuurityyleihin. Luvussa 4 esitellään perinteisten monen sivun sovellusten sekä modernien yhden sivun sovellusten toimintaa. Luvussa 5 käsitellään tärkeimmät perusperiaatteet kolmesta web-sovellusten toteutukseen käytetystä teknologiasta. Modernisoinnin menetelmiä käydään läpi luvussa 6 ensin yleisellä tasolla ja lopulta muutamaan menetelmään yksityiskohtaisemmin perehtyen. Luvussa 7 pohditaan tarkemmin tiedonhaussa ilmenneitä haasteita sekä esitellään työssä saavutetut tulokset ja johtopäätökset. Luvussa 8 kootaan yhteen työn pääkohdat.

2. PERUSKÄSITTEET

Web-sivut muodostuvat HTML-merkintäkielen, CSS-tyylien ja JavaScript-ohjelmointikielen kokonaisuutena. URI-tunnisteen avulla sovellus tunnistaa haettavan web-sivun tai muun resurssin. Resurssi, kuten tietyn tiedon sisältämä data, voidaan siirtää esimerkiksi JSON-muodossa. Resurssien siirto sekä muu palvelimen ja selaimen välinen kommunikaatio puolestaan perustuu HTTP-protokollaan. Tässä luvussa esitellään joukko web-sovellusten toiminnan kannalta olennaisia peruskäsitteitä, jotka tulevat esiintymään myöhemmissä tekstiluvuissa.

2.1 HTML JA CSS

HTML-merkintäkieli (HyperText Markup Language) määrittelee web-sivun ja sen sisällön rakenteen ja merkityksen [32]. Web-selain jäsentää merkintäkielen DOM-puuksi (Document Object Model), joka on muistissa säilytettävä kuvaus HTML-dokumentista. Alkujaan HTML suunniteltiin tieteellisten tekstidokumenttien kuvaamiseen, mutta sen tarjoamat ominaisuudet ovat merkittävästi laajentuneet uusien versioiden myötä. Uusin käytössä oleva versio on HTML5. [33]

HTML muodostuu joukosta elementtejä, jotka rajaavat dokumentin sisällöstä tietyllä tavalla käyttäytäytyviä tai ulkonäöltöön tietynlaisia osioita. Elementtejä ovat esimerkiksi linkit, kuvat ja otsikot. [32] Elementti muodostuu yleensä aloittavasta ja sulkevasta tagista sekä niiden väliin jäävästä sisällöstä. Sisältö voi koostua tekstin lisäksi myös toisista elementeistä. [33] Elementteihin voidaan liittää attribuutteja, jotka antavat lisäinformaatioa elementtiin liittyen. Attribuutit eivät kuitenkaan sellaisenaan ole näkyvillä varsinaisessa käyttäjälle esitettävässä sisällössä. [32]

Ohjelmassa 2.1 esitetään yksinkertainen HTML-dokumentti. `Html`-elementti sisältää kaikki muut web-sivun elementit. Sille on asetettu attribuutti `lang`, joka kertoo dokumentin kielen. `Head`-elementtiin asetetaan tyypillisesti sisältöä, jonka ei haluta näkyvän suoraan sivun sisällössä, kuten sivun nimi ja metatiedot. `Body`-elementti sen sijaan rajaa sisäänsä käyttäjälle näytettävän sisällön. Tässä tapauksessa sivulla on otsikkoelementti `h1`, sekä linkin `a` sisältävä tekstikappale `p`. [32, 33]

CSS (Cascading Style Sheets) on tyylikieli, jonka avulla HTML-dokumentin elementeil-

Ohjelma 2.1. *Yksinkertainen HTML-dokumentti.*

```
1  <!DOCTYPE html>
2  <html lang="fi">
3    <head>
4      <meta charset="utf-8">
5      <title>Esimerkkisivu</title>
6    </head>
7    <style>
8      h1 {
9        font-style: italic;
10     }
11     .teksti {
12       color: red;
13     }
14   </style>
15   <body>
16     <h1>Otsikko</h1>
17     <p class="teksti">Tekstikappale, jossa <a href="">linkki</a>.</p>
18     <!-- kommentti -->
19   </body>
20 </html>
```

le voidaan asettaa tyylejä ja siten vaikuttaa niiden ulkonäköön. CSS muodostuu valitsimista ja valitsimille määritellyistä säännöistä. Valitsimilla poimitaan HTML-dokumentista elementtejä, joihin valitsimen ominaisuudet vaikuttavat. [21] Ohjelmaan 2.1 on liitetty `style`-elementti, joka sisältää web-sivun tyylit. Yleensä CSS sijoitetaan kuitenkin omaan tyylitiedostoonsa. `H1`-valitsimella valitaan HTML-dokumentista `h1`-elementti, jolle määritellään elementin sisältämän tekstin tyyliin vaikuttava sääntö. Tässä tapauksessa otsikon teksti kursivoidaan. Vastaavasti `.teksti`-valitsin valitsee dokumentista kaikki `class`-attribuutilla `teksti` merkityt elementit, ja määrittää niiden fontin väriksi punaisen.

2.2 URI

URI (Uniform Resource Identifier) on tiettyyn syntaksiin perustuva merkkijono, jota käytetään resurssien tunnistamiseen. Resurssi on mikä tahansa asia tai olio, joka voidaan tunnistaa URI:n avulla. Tyypillisiä resursseja ovat esimerkiksi sähköiset dokumentit, kuvat, palvelut tai useammasta resurssista koostuvat kokoelmat. Usein resurssi on saatavilla internetissä, mutta tämä ei ole välttämätöntä. [15]

URI mahdollistaa tunnisteiden yhdenmukaisuuden. Toisistaan poikkeavat resurssien tunnisteet ja tyypiltään erilaisiin resursseihin viittaavat tunnisteet voidaan esittää yhtenäisellä tavalla. Tunnisteen on sisällettävä kaikki sellainen informaatio, joka on tarpeen resurssin tunnistamiseen ja sen erottamiseen muista resursseista. Tunnisteen ei tarvitse ottaa kan-

taa siihen, millä tavoin tunnisteen identifioima resurssi haetaan. [15]

URI-tunniste voi toimia resurssin osoitteena, nimenä tai saman aikaisesti osoitteena ja nimenä. URN (Uniform Resource Name) on URI, jonka on oltava maailmanlaajuisesti uniikki ja pysyvä, vaikka sen nimeämää resurssia ei olisi enää olemassa tai saatavilla. URL (Uniform Resource Locator) puolestaan sisältää resurssin tunnistamisen lisäksi tarvittavan tiedon resurssin paikantamiseen, kuten sen sijainnin tietoverkossa. Lisäksi URL kuvaa ensisijaisen mekanismin resurssin hakemiseen. [15] URL-tunnisteesta käytetäänkin usein nimitystä URL-osoite.

URL-tunniste voidaan jakaa viiteen toisistaan erotettavaan osaan. Tarkastellaan esimerkiksi Tampereen yliopiston tarjoamien opintojaksojen verkkosivun lyhynnettyä URL-osoitetta: `https://www.tuni.fi/opintojaksot/?year=2021&location=hervanta`. Ensimmäinen osa (`https`), osoittaa protokollan, jota selaimen tulee käyttää resurssia pyytäessään. Tässä tapauksessa käytössä on HTTP-protokollan salattu versio HTTPS. Toinen osa (`www.tuni.fi`), sisältää verkkosivun domainin ja kertoo, miltä palvelimelta resurssia pyydetään. Tämä voitaisiin esittää myös IP-osoitteen muodossa. Domainin jälkeen merkitään käytetty portti. Jos porttia ei erikseen ole annettu, käytetään oletusporttia. HTTP-protokollaa käytettäessä oletusportti on 80 ja HTTPS-protokollaa 443. [92]

URL-tunnisteen kolmas osa (`/opintojaksot`) muodostuu polusta, joka osoittaa resurssin sijainnin tietyllä palvelimella. Polku voi kuvata suoraan tiedoston absoluuttisen sijainnin. Nykyään kuitenkin käytetään usein abstraktiota, josta sijaintia ei voida päätellä. Polun jälkeen merkitään tarvittavat parametrit (`?year=2021&location=hervanta`). Parametrit ovat avain-arvo-pareja, jotka erotetaan toisistaan &-merkillä. Parametrien perusteella palvelin suorittaa resurssille lisätoimenpiteitä ennen sen palauttamista. Esimerkiksi tässä tapauksessa parametrejä käytetään todennäköisesti opintojaksojen rajaamiseen. Viimeisenä osana URL voi sisältää ankkurin, joka osoittaa esimerkiksi tiettyyn elementtiin HTML-dokumentissa. Tällöin selain kykenee vierittämään näkymän suoraan URL-osoitteessa merkittyyn elementtiin. [92]

2.3 HTTP

HTTP (Hypertext Transfer Protocol) on sovelluskerroksen protokolla, johon merkittävä osa internetissä tapahtuvasta tiedonsiirrosta perustuu [2, 47]. Yleensä HTTP toimii TCP-protokollan päällä, mutta määritelmän mukaan mitä tahansa luotettavaa kuljetuskerroksen protokollaa voidaan käyttää [47]. Luotettava protokolla ei hukkaa siirrettävää dataa tai ainakin antaa virheen näin tapahtuessa [2]. Internetissä HTTP:n ensimmäinen versio otettiin käyttöön 1990-luvun alkupuolella, minkä jälkeen siihen on lisätty ominaisuuksia uudemmissa versiossa [2, 47].

HTTP:n toiminta perustuu asiakas- ja palvelinkomponenttien väliseen, erillisistä viesteis-

tä muodostuvaan kommunikaatioon. Asiakas voi olla mikä tahansa ohjelma, joka muodostaa yhteyden palvelinpyyntöjen (engl. *request*) lähettämiseen. Palvelin puolestaan on ohjelma, joka hyväksyy yhteyksiä ja palvelee asiakkaan tekemiä pyyntöjä palauttamalla vastauksia (engl. *response*). Ohjelma voi samanaikaisesti toimia sekä asiakkaana että palvelimena. [47] Asiakas voi tehdä palvelimelle useita pyyntöjä avaamalla jokaista perättäistä pyyntöä varten oman yhteyden, uudelleen käyttämällä vanhaa tai avaamalla kerralla useampia yhteyksiä [2]. HTTP on tilaton protokolla, joten kaksi perättäistä pyyntöä eivät ole tietoisia toisistaan [47]. Yhteisiä tietoja sisältäviä istuntoja (engl. *session*) voidaan kuitenkin muodostaa HTTP-evästeiden avulla [2].

Tyypillisesti HTTP-protokollaa käytetään resurssien kuten HTML-dokumenttien hakemiseen tai sisällön lähettämiseen palvelimelle [2]. Yleensä asiakas, kuten web-selain, aloittaa HTTP-kommunikaation lähettämällä pyynnön palvelimelle. Palvelin käsittelee pyynnön soveltamalla sitä haluttuun resurssiin ja palauttaa vastauksen. [47] Esimerkiksi näyttääkseen web-sivun käyttäjälle selain lähettää palvelimelle pyynnön HTML-dokumentin hakemiseksi. Selain käsittelee palvelimen vastauksena saadun dokumentin ja tekee tarpeen mukaan uusia pyyntöjä hakeakseen skriptejä, CSS-tiedostoja sekä muita sivun sisältöjä kuten kuvia ja videoita. Selain kasaa lopullisen dokumentin haetuista dokumentin osista. Selaimessa suoritetut skriptit voivat hakea myöhemmin lisää resursseja, joiden mukaan selain päivittää dokumenttia. [2]

Asiakkaan tekemä pyyntö sisältää käytetyn metodin, URI-tunnisteen, HTTP:n version, valinnaisia lisätietoja sisältäviä otsikkotietoja ja mahdollisen sisällön (engl. *body content*) [2, 47]. Metodi määrittelee operaation, jonka asiakas haluaa suorittaa [2]. Esimerkiksi GET, POST, PUT ja DELETE ovat yleisesti käytettyjä metodeja. GET palauttaa minkä tahansa resurssin, joka tunnistetaan pyynnön sisältämällä URI-tunnisteella. POST lisää tai muuttaa URI:n osoittaman resurssin dataa. Uusi data lähetetään pyynnön sisällössä. PUT puolestaan luo kokonaan uuden resurssin halutulla URI-tunnisteella tai päivittää olemassa olevaa resurssia. DELETE poistaa URI-tunnisteen mukaisen resurssin. [47]

Palvelin vastaa viestillä, joka sisältää HTTP:n version, onnistumista tai virhettä kuvaavan statuskoodin, palvelimen tietoja ja metatietoja sisältäviä otsikkotietoja sekä mahdollisen sisällön, joka voi muodostua esimerkiksi haetusta resurssista [2, 47]. Statuskoodit sisältävät kolminumeroisen luvun ja muutaman sanan kuvauksen statuksesta. Samalla numerolla alkavat statuskoodit viittaavat saman tyyppisiin vastauksiin. Esimerkiksi luvulla kaksi alkavat liittyvät onnistuneeseen pyyntöön, neljällä alkavat asiakkaan virheeseen ja viidellä alkavat palvelimen virheeseen. [47]

2.4 JavaScript

JavaScript on korkean tason dynaaminen, tulkattava ja oliosuuntautunut ohjelmointikieli, joka soveltuu hyvin myös funktionaaliseen ohjelmointiin [22, 25]. JavaScript julkaistiin

vuonna 1997 Netscapen web-selaimessa, ja vuoden 2015 kuudennen version jälkeen uusi versio on julkaistu vuosittain [22]. Lisensisointisyistä johtuen standardisoitu kieli on nimetty ECMAScriptiksi, mutta käytännössä kieltä kutsutaan yleensä kuitenkin JavaScriptiksi [25]. Vaikka JavaScript muistuttaa nimeltään Javaa, ei sillä syntaksin samankaltaisuuden lisäksi ole yhteyksiä Java-kieleen [22].

Alun perin JavaScript suunniteltiin internetin skriptauskieleksi, mutta ajan kuluessa siitä on kehittynyt vakavasti otettava yleiskäyttöinen ohjelmointikieli, joka soveltuu myös suuren mittakaavan ohjelmistoprojekteihin. Nykyään JavaScript onkin yksi käytetyimmistä ohjelmointikielistä. Tärkeä virstanpylväs JavaScriptin kehityksessä saavutettiin vuoden 2015 kuudennessa versiossa, jossa kieleen lisättiin esimerkiksi luokat ja moduulit. [22, 25] Strict-tilan avulla kielen vanhempia ominaisuuksia voidaan rajoittaa esimerkiksi turvallisuuden nostamiseksi, helposti virheitä aiheuttavien ominaisuuksien välttämiseksi tai virheiden käsittelyn parantamiseksi [22]. Taaksepäin yhteensopivuuden varmistamiseksi alkuaikojen heikoiksi tai huonoiksi miellettyjä ominaisuuksia ei ole haluttu kokonaan poistaa [25].

JavaScript sisältää itsessään varsin suppean rajapinnan esimerkiksi lukujen, tekstien ja taulukoiden kanssa työskentelyyn. Muun toiminnallisuuden, kuten tiedon syöttämisen ja sen tulostamisen, tiedostojen käsittelyn tai tietoverkkojen käyttämisen mahdollistamiseksi JavaScript vaatii isäntäympäristön (engl. *host environment*), joka tarjoaa tarvittavat oliot ja rajapinnat. Alkuperäinen ja yleisin isäntäympäristö on web-selain, joka antaa JavaScript-koodille mahdollisuuden käsitellä esimerkiksi käyttäjän hiirellä tai näppäimistöllä suorittamia toimintoja ja muita ympäristön tapahtumia, kuten lomakkeen lähetyksen. [22, 25] Lisäksi JavaScript-koodin avulla voidaan dynaamisesti muokata HTML-dokumentin DOM-puuta ja tehdä HTTP-pyyntöjä [25]. Selainympäristö sisältää joukon olioita, jotka esittävät esimerkiksi tekstialueita, menuja, ikkunoita, selaimen historiaa ja evästeitä [22]. Käytännössä kaikki modernit web-selaimet eri alustoilla sisältävät JavaScript-tulkin [25]. JavaScriptia voidaan nykyään käyttää myös palvelinpuolen ohjelmointiin. Node.js-ympäristö mahdollistaa esimerkiksi tiedostojen lukemisen ja kirjoittamisen, datan lähettämisen ja vastaanottamisen tietoverkossa sekä HTTP-pyyntöjen tekemisen ja palvelemisen JavaScriptin avulla. [48, 25]

JavaScript perustuu vahvasti olioihin, ja JavaScript-ohjelmaa voidaankin kuvata joukkona toistensa kanssa kommunikoivia olioita. Olio muodostuu kokoelmasta ominaisuuksia (engl. *properties*), jotka sisältävät ominaisuuden nimen ja sen arvon. Lisäksi jokaisella ominaisuudella on attribuutteja, jotka määrittelevät miten ominaisuutta voidaan käyttää. Esimerkiksi jos ominaisuuden `Writable`-attribuutin arvo on `false`, ei ominaisuuden arvoa ole mahdollista muuttaa. Ominaisuuksia voidaan dynaamisesti lisätä olioihin asettamalla ominaisuuksille arvoja. Ominaisuuksien arvot ovat joko primitiivisiä arvoja tai olioita. Primitiiviset arvot ovat JavaScriptin sisäänrakennettuja tyyppejä, kuten `Undefined`, `Boolean`, `Number` tai `String`. JavaScriptissa myös funktiot ovat olioita, joten ne voidaan

asettaa ominaisuuden arvoksi. Tällöin funktiota kutsutaan metodiksi. [22] Tyypillinen olio muodostuu järjestämättömästä kokoelmasta nimettyjä arvoja. JavaScriptissa on määritelty myös erityisiä tiettyihin käyttötarkoituksiin suunnattuja olioita, kuten `Array`, `Map`, `Error` ja `Date`. JavaScript käyttää automaattista roskienkeruuta, joten yleensä olioiden tuhoamisesta ja muistin vapauttamisesta ei erikseen tarvitse huolehtia. [25]

Toisin kuin monissa muissa kielissä, JavaScriptissa oliot eivät pohjimmiltaan perustu luokkiin. Olioita voidaan luoda useilla eri tavoilla, kuten rakentajilla (engl. *constructor*). Rakentajafunktio luo olion ja suorittaa koodin, joka alustaa olion asettamalla alkuarvot olion ominaisuuksiin. Rakentajat sisältävät `prototype`-ominaisuuden, joka mahdollistaa prototyyppiin perustuvan periytymisen ja ominaisuuksien jakamisen. Jokaisella rakentajan luomalla oliolla on viite rakentajan prototyyppiominaisuuden arvoon. Tätä kutsutaan olion prototyypiksi, ja toisiinsa prototyyppiviitteen kautta linkittyneitä olioita prototyyppiketjuksi. Olio perii prototyypiltään kaikki ne ominaisuudet, joita se ei itse suoraan sisällä. [22]

2.5 JSON

JSON (JavaScript Object Notation) on kevyeksi ja ohjelmointikielten välillä siirrettäväksi suunniteltu tekstipohjainen tiedonvälityksen muoto [17]. Sitä käytetään yleisesti datan siirtämiseen web-sovelluksissa esimerkiksi lähetettäessä dataa asiakkaalta palvelimelle tai päinvastoin [93]. JSON:in syntaksi perustuu JavaScriptin olioihin, mutta on laajuudeltaan suppeampi. Kaikkia olioiden arvoja ei siis suoraan kyetä esittämään JSON-muodossa. JavaScriptissa oliot voidaan kuitenkin muuntaa JSON:iksi ja JSON:ista takaisin olioiksi [25]. JavaScript-taustasta huolimatta JSON ei ole riippuvainen yksittäisistä ohjelmointikielistä, vaan sitä voidaan käyttää viestintään myös ohjelmointikielten välillä. [17]

JSON muodostuu tekstistä, joka voi sisältää merkkijonoja, numeroita, kolmea eri nimiliteraalia (`false`, `true`, `null`) ja kuutta tiettyyn tarkoitukseen varattua merkkiä (: , [ ] { }). Tyyppejä on mahdollista esittää kuusi erilaista: primitiivityypit merkkijono, numero, totuusarvo ja `null`, sekä jäsennellyt (engl. *structured*) tyypit olio ja taulukko. Olio on rakenne, jossa aaltosulkujen väliin asetetaan järjestämätön kokoelma nimi-arvo-pareja. Nimi muodostuu merkkijonosta, jonka tulisi olion sisällä olla uniikki. Arvo voi muodostua mistä tahansa JSON:in sisältämästä tyypistä. Nimi ja arvo erotetaan toisistaan kaksoispisteellä, ja nimi-arvo-parit pilkuilla. Taulukko puolestaan koostuu tyypillisesti järjestetystä kokoelmasta arvoja, jotka ovat erotettu toisistaan pilkulla. Taulukon alku- ja loppu rajataan hakasulkeilla. [17]

Ohjelmassa 2.2 esitetään yksinkertainen JSON-tiedosto. Tiedosto sisältää olion, joka muodostuu `nimi`- ja `teokset`-ominaisuuksista. `Nimi`-ominaisuus on arvoltaan tavallinen merkkijono. `Teokset` puolestaan sisältää taulukon, jossa on kaksi olioista muodostuvaa arvoa. Kumpikin taulukon olioista sisältää viisi ominaisuutta. `Kirjoittajat`-ominaisuuden arvo on molemmissa olioissa taulukko. Vastaavasti `vuosi` on tyypiltään numero, `nimi`

Ohjelma 2.2. *Yksinkertainen JSON-tiedosto.*

```
1  {
2    "nimi": "Kirjaston kokoelma",
3    "teokset":
4      [
5        {
6          "kirjoittajat": ["Roy Fielding", "Richard Taylor"],
7          "vuosi": 2002,
8          "nimi": "Principled Design of the Modern Web Architecture",
9          "issn": "1533-5399",
10         "saatavilla": false
11       },
12       {
13         "kirjoittajat": ["Mark Richard"],
14         "vuosi": 2015,
15         "nimi": "Software architecture patterns",
16         "issn": null,
17         "saatavilla": true
18       }
19     ]
20 }
```

merkkijono ja `saatavilla` totuusarvo. `Issn`-ominaisuuden arvo kuitenkin puuttuu jälkimmäiseltä oliolta, ja ominaisuudelle on annettu arvoksi `null`.

3. WEB-SOVELLUSTEN OHJELMISTOARKKITEHTUURI

Sovelluksen ohjelmistoarkkitehtuuria voidaan kuvata esimerkiksi järjestelmän pohjapiirrokseksi tai korkean tason etenemissuunnitelmaksi järjestelmän kehittämiseksi. Molemmat edellä mainituista kuvauksista jäävät kuitenkin hyvin epämääräisiksi ja tulkinnanvaraisiksi. Ohjelmistoarkkitehtuurin käsite onkin varsin haastavaa määritellä täsmällisesti ja yksikäsitteisesti. [27] Kokonaiskuvan muodostamiseksi esitetään seuraavaksi kaksi hieman toisistaan poikkeavaa määritelmää ohjelmistoarkkitehtuurista.

Fieldingin [24] mukaan ohjelmistoarkkitehtuuri on abstraktio järjestelmän ajonaikaisista elementeistä jossain järjestelmän toimintavaiheessa. Toimintavaiheita ovat esimerkiksi käynnistys, normaali ajonaikainen prosessointi ja sammutus. Abstraktiolla tarkoitetaan tässä yhteydessä järjestelmän ja sen elementtien sisäisten yksityiskohtien piilottamista. Ohjelmistoarkkitehtuuri kuvaa abstraktiota järjestelmän ajonaikaisesta käyttäytymisestä ja eroaa siten ohjelmiston rakenteesta, joka on staattisen lähdekoodin ominaisuus. [24]

Monimutkainen järjestelmä sisältää useita abstraktiotasoja, joilla voi olla omat arkkitehtuurinsa. Arkkitehtuuri esittää järjestelmän käyttäytymisen abstraktiota kyseisellä tasolla siten, että arkkitehtuurisia elementtejä rajoittavat rajapinnat, joita ne tarjoavat muille saman tason elementeille. Elementit voivat koostua alielementeistä, joilla on puolestaan omat arkkitehtuurinsa. [24] Arkkitehtuuri määrittelee, miten järjestelmän elementit tunnistetaan ja jaetaan, miten elementit ovat vuorovaikutuksessa toistensa kanssa muodostaakseen järjestelmän, millaista kommunikaatiota vuorovaikutukseen tarvitaan ja vuorovaikutukseen käytettävät protokollat [23].

Arkkitehtuurisia elementtejä ovat komponentit, liittimet ja data. Komponentti on abstrakti yksikkö, joka koostuu ohjelmakoodista ja sisäisestä tilastaan. Sen tehtäviä ovat laskenta ja datan muuntaminen toiseen muotoon. Komponentti määräytyy sen mukaan, mitä palveluita se tarjoaa muille komponenteille rajanpintansa kautta. Komponenttien välisestä kommunikaatiosta, koordinoinnista ja yhteistyöstä vastaavaa abstraktia mekanismia kutsutaan liittimeksi. Liitin siirtää dataa komponenttien rajapintojen välillä muuttamatta datan sisältöä tai muotoa. Data määritellään informaatioelementiksi, jonka komponentti ottaa vastaan tai joka siirtyy komponentista liittimen siirtämänä. Data ei sisällä komponentin sisäiseksi tarkoitettua informaatiota. [24]

Arkkitehtuuriset elementit muodostavat konfiguraation, jota rajoittavat elementtien väliset suhteet. Konfiguraatio kuvaa elementtien välisten suhteiden rakennetta tietyllä aikavälillä järjestelmän ajonaikana, määrittäen näin järjestelmän arkkitehtuurin. Tällöin muodostuu kokonaisuus, jolla on haluttu joukko arkkitehtuurisia ominaisuuksia. Arkkitehtuuriset ominaisuudet muodostuvat elementtien suhteen tehdyistä valinnoista ja järjestelyistä. Ominaisuudet ovat seurausta arkkitehtuurin rajoitteista ja voivat olla tyypiltään joko toiminnallisia tai laadullisia. Arkkitehtuurisia ominaisuuksia ovat esimerkiksi suorituskyky, skaalautuvuus ja muokattavuus. [24]

Ford ja Richards [27] määrittelevät ohjelmistoarkkitehtuurin koostuvan järjestelmän rakenteesta, arkkitehtuurisista ominaisuuksista, arkkitehtuurisista päätöksistä ja suunnitteluperiaatteista. Lähestymistapa eroaa Fieldingin [24] määritelmästä, jossa korostuvat erityisesti abstraktion käsite, ajonaikaisuus ja jako arkkitehtuurisiin elementteihin. Fordin ja Richardsin [27] määritelmässä järjestelmän rakenne viittaa valittuihin arkkitehtuurityyleihin, joilla järjestelmä toteutetaan. Tässä tapauksessa arkkitehtuurityyli kuvaa yleistä rakennetta siitä, miten järjestelmän koodi on jaettu esimerkiksi kerroksiin tai erillisiksi palveluiksi, ja kuinka lähdekoodi on vuorovaikutuksessa tietovaraston kanssa. Arkkitehtuurityyli kattaa joukon sekä hyödyllisiä että haitallisia arkkitehtuurisia ominaisuuksia.

Arkkitehtuuriset ominaisuudet voivat olla joko toiminnallisia tai laadullisia ja ne vaikuttavat järjestelmän rakenteeseen. Ne kuvaavat järjestelmän ominaisuuksia, jotka eivät suoraan liity järjestelmän sovellusalueeseen. Fordin ja Richardsin [27] määritelmä arkkitehtuurisista ominaisuuksista on hyvin samankaltainen Fieldingin [24] kanssa. Ford ja Richards [27] korostavat kuitenkin arkkitehtuuristen ominaisuuksien vaikuttavan merkittävästi järjestelmän onnistuneeseen toteutukseen ja täten määrittelevän järjestelmän onnistumiskriteerit.

Arkkitehtuuriset päätökset määrittelevät säännöt, joiden mukaan järjestelmä tulee rakentaa. Esimerkki tällaisesta säännöstä voisi olla, että vain tietyt järjestelmän tasot saavat olla yhteydessä tietokantaan. Mikäli tilanteessa ei voida ottaa huomioon kaikkia olennaisia vaihtoehtoja tai olosuhteita, voidaan arkkitehtuuristen päätösten sijaan käyttää suunnitteluperiaatteita. Suunnitteluperiaatteet ovat ohjeita ja suosituksia, eivätkä ne rajoita järjestelmän rakennetta yhtä vahvasti kuin arkkitehtuuriset päätökset. Suunnitteluperiaate voi antaa suosituksen toivotusta tavasta esimerkiksi komponenttien välisen kommunikaation toteuttamiseen. Kehittäjän vastuulle jää kuitenkin yksityiskohtien, kuten viestiprotokollan valinta. [27]

Ohjelmistoarkkitehtuureja voidaan kuvata niiden toteuttamilla arkkitehtuurityyleillä. Fieldingin [24] mukaan joukkoa arkkitehtuurisia rajoitteita ja ominaisuuksia, jotka vaikuttavat elementtien välisiin suhteisiin ja elementtien rooleihin, kutsutaan arkkitehtuurityyliksi. Tyylien avulla arkkitehtuureja jaetaan kategorioihin ja määritellään niiden yhteisiä piirteitä. Arkkitehtuurityylin rajoitteet pätevät kaikkiin niihin arkkitehtuureihin, jotka käyttävät kyseistä tyyliä. Lisäksi yksi arkkitehtuuri voi koostua useammasta eri tyylistä. [24] Web-

sovelluksissa yleisesti käytettyjä arkkitehtuurityylejä ovat esimerkiksi kerros-, asiakaspalvelin-, REST- ja MV*-arkkitehtuurityylit. Joissain lähdemateriaaleissa arkkitehtuurityyli ja arkkitehtuurimalli (engl. *architecture pattern*) erotetaan toisistaan, mutta jako ei ole selkeä tai vakiintunut. Koska näiden käsitteiden erottaminen ei tämän työn tavoitteiden kannalta ole olennaista, viitataan käsitteellä arkkitehtuurityyli sekä tyyleihin että malleihin.

3.1 Kerrosarkkitehtuurityyli

Kerrosarkkitehtuurityyli muodostuu horisontaalisista kerroksista, joista jokaisella on tietty roolinsa ja tehtävänsä järjestelmässä. Kerrosten määrät ja tyypit vaihtelevat toteutuksesta riippuen, mutta tyypillisesti kerrosarkkitehtuurityyli sisältää neljä kerrosta: esityskerroksen, sovelluskerroksen, tallennuskerroksen ja tietokannan. [62] Nämä kerrokset on esitetty kuvassa 3.1. Esimerkiksi sovellus- ja tallennuskerros voidaan kuitenkin yhdistää yhdeksi kerrokseksi tai sovelluskerroksesta voidaan erottaa palvelinpyynnöistä vastaava palvelukerros omaksi tasokseen [62] [71].

Esityskerros vastaa datan muotoilusta ja näyttämisestä käyttäjälle. Sovelluskerros suorittaa tarvittavan datan prosessoinnin saapuneeseen palvelinpyyntöön liittyen. Sovelluskerros pyytää datan tallennuskerrokselta, joka vastaa kommunikaatiosta tietokannan kanssa. [62] Kerrosten ei tarvitse tietää muiden kerrosten sisäisestä toteutuksesta. Esimerkiksi sovelluskerroksen ei tarvitse tuntea yksityiskohtia siitä, miten data haetaan, missä sitä säilytetään tai kuinka se näytetään käyttöliittymässä. Näin ollen muutokset yhteen kerrokseen eivät aiheuta muutoksia muihin kerroksiin. Tätä kutsutaan vastuiden erottamisen periaatteeksi (engl. *separation of concerns*). [27]

Yksittäinen kerros tarjoaa palveluita sitä ylemmälle kerroksella ja käyttää itse vastaavasti alemman kerroksen palveluita [24]. Tällöin kerrokset ovat suljettuja, jolloin pyynnön on kuljettava kaikkien tasojen läpi ylhäältä alaspäin. Samoin pyynnön on palattava jokaisen

Kuva 3.1. *Kerrosarkkitehtuurityylin rakenne ja pyynnön kulku kerrosten läpi.*

kerroksen kautta. Pyynnön kulkua kerrosten läpi on kuvattu nuolilla kuvassa 3.1. Erikoistapauksissa kerrokset voivat myös olla avoimia, jolloin esimerkiksi esityskerros voi suoraan pyytää dataa tallennuskerrokselta sovelluskerroksen ohi. Tämä nopeuttaa pyynnön prosessointia, mutta luo riippuvuuksia kerrosten välille ja haasteita kerrosten vastuiden erottamiseen. [62]

3.2 Asiakas-palvelin-arkkitehtuurityyli

Tyypillisesti web-sovellusten toteutuksessa on sovellettu asiakas-palvelin-arkkitehtuurityyliä. Tällöin järjestelmä pyritään jakamaan kahteen erilliseen osaan, jotka vastaavat tietyistä toiminnallisuuksista. Yleensä järjestelmästä erotetaan datan näyttämiseen erikoistunut asiakasosa ja datan käsittelystä vastaava palvelin. [27] Palvelin tarjoaa joukon erilaisia palveluita ja odottaa asiakkaan tekemiä palvelinpyyntöjä. Palvelinpyynnön saapuessa palvelin joko hyväksyy tai hylkää pyynnön, suorittaa tarvittavan prosessoinnin ja lähettää vastauksen takaisin asiakkaalle. [24]

Asiakkaan ja palvelimen lisäksi asiakas-palvelin-arkkitehtuurityylissä voidaan hyödyntää välittäjäkomponentteja, kuten välityspalvelinta (engl. *proxy*) ja yhdyskäytävää (engl. *gateway*). Välityspalvelin ottaa vastaan asiakkaan palvelinpyynnöt ja välittää ne eteenpäin varsinaisen prosessoinnin suorittavalle web-palvelimelle. Välityspalvelin voi myös tarvittaessa tehdä muutoksia palvelinpyyntöön. Yhdyskäytävä näyttää asiakkaalle normaalilta palvelimelta. Se kuitenkin vain välittää saapuneet pyynnöt eteenpäin muille palvelimille, eikä itse suorita muuta laskentaa. Välittäjäkomponenttien avulla voidaan toteuttaa kuormantasausta tai lisätä erilaisia turvallisuusominaisuuksia. [24]

Web-sovelluksissa asiakkaana toimii yleensä web-selain, joka ottaa yhteyttä web-palvelimeen. Palvelin puolestaan on tarvittaessa yhteydessä tietokantaan. [27] Web-palvelin tarjoaa palvelua tyypillisesti useammalle kuin yhdelle asiakkaalle [24]. Selainten ja asiakaslaitteiden ominaisuudet vaihtelevat merkittävästi riippuen siitä, onko kyseessä esimerkiksi työasema tai mobiililaite [70]. Asiakas- ja palvelinkomponentteja voidaan muuttaa toisistaan riippumatta siten, että niiden välinen rajapinta pysyy samana [24]. Näin ollen selainten ja asiakaslaitteiden toisistaan eroavat ominaisuudet, kuten näytön koko ja suunta, voidaan ottaa huomioon asiakaskomponentin toteutuksessa. Asiakas- ja palvelinkomponenttien erottaminen toisistaan mahdollistaa myös niiden toteuttamisen eri teknologioilla. Nykyisin asiakas on yleensä toteutettu JavaScriptilla olemassa olevia kirjastoja ja rajapintoja hyödyntäen. Palvelinpuolella teknologiat vaihtelevat enemmän, mutta myös palvelin on mahdollista toteuttaa JavaScriptilla esimerkiksi Node.js-ajoympäristön avulla [48].

Asiakas-palvelin-arkkitehtuurityyli ei perusmuodossaan ota kantaa siihen, miten sovelluksen tila jaetaan palvelimen ja asiakkaan välillä [24]. Aiemmin web-sovellukset olivat melko staattisia ja sisälsivät vain vähän dynaamista sisältöä. Palvelinlaitteet olivat suhteellisen nopeita verrattuna JavaScriptin suorittamiseen selaimessa. Lisäksi selaimet kärsivät

lukuisista yhteensopimattomuusongelmista, joiden ratkaisuun oli tarjolla vain vähäinen määrä kirjastoja ja tekniikoita. Täten oli luontevaa sijoittaa suurin osa web-sovelluksen toimintalogiikasta palvelimelle. [70]

Web-sovellukset ovat kuitenkin muuttuneet yhä interaktiivisemmiksi ja monipuolisemmiksi. Toisaalta selainten ja JavaScript-ohjelmointikielen kehitys mahdollistavat aiempaa raskaampien ja monimutkaisempien ohjelmien suorittamisen selaimessa. Näin ollen web-sovellusten toimintalogiikkaa on siirretty palvelimelta asiakkaskomponenttiin. Esimerkiksi Kuuskerin ja Mikkosen [37] mukaan moderneja web-sovelluksia toteutettaessa mahdollisimman suuri osa käyttäjän suorittamista toiminnoista tulisi käsitellä asiakaskomponentissa, jolloin järjestelmän reagointinopeutta voidaan kasvattaa. Palvelimen vastuulle jätettäisiin vain sellaiset toiminnot, joita asiakas ei kykene suorittamaan [37].

Kuuskeri ja Mikkonen [37] suosittelevat, että palvelimen tulisi vastata kaikesta sellaisesta datasta, jonka on säilyttävä myös sovelluksen sulkemisen jälkeen. Modernit web-selaimet tarjoavat kuitenkin monia ratkaisuja myös pysyvän datan tallentamiseen [52]. Mikäli tallennettavaa dataa on paljon tai sen säilyttämistä ei datan luonteesta johtuen voida jättää asiakkaan vastuulle, on datan tallentaminen edelleen varsin perusteltua suorittaa palvelimella. Myös jaettu data, joka on useamman asiakkaan käytössä samanaikaisesti, tulee säilyttää palvelimella. [37]

Asiakkaiden välinen vuorovaikutus toteutetaan palvelimen välityksellä. Lisäksi palvelin voi olla yhteydessä muihin palveluihin esimerkiksi tarvittavan datan hakemiseksi. Koska käyttäjä voi sulkea selaimen kesken operaation suorituksen, on joidenkin luotettavuutta vaativien toimintojen suoritus syytä hoitaa palvelimella. Samoin tiettyyn aikaan suoritettavat ja aikataulutetut tehtävät ovat palvelimen vastuualuetta. Mikäli operaation suorittaminen vaatii raskasta laskentaa, voidaan se optimoida paremmin palvelimella, kun käytössä oleva laitteisto tunnetaan. Lisäksi on mahdollista käyttää tehokkaampaa ohjelmointikieltä JavaScriptin sijaan. [37]

3.3 REST

REST (Representational State Transfer) on arkkitehtuurityyli, joka muodostuu tiettyjä ominaisuuksia sisältävistä ja muihin arkkitehtuurityyleihin perustuvista arkkitehtuurisista rajoitteista [23]. Tavoiteltuja ominaisuuksia ovat esimerkiksi käyttöliittymän siirrettävyys, skaalautuvuus, komponenttien itsenäisyys ja virhesietoisuus [23, 70]. REST käyttää hyväkseen HTTP-protokollaa ja olemassa olevia internetin rakenteita [34]. Nimi “representational state transfer” perustuu näkemykseen hyvin suunnitellun web-sovelluksen toiminnasta. Web-sovellus toimii virtuaalisena tilakoneena, jonka läpi käyttäjä navigoi sovelluksen tarjoamien linkkien tai lomakkeiden kautta. Sovellus vastaa linkin painallukseen tai lomakkeen lähetykseen lähettämällä käyttäjälle sovelluksen seuraavan tilan esityksen, jolloin sovelluksen tila muuttuu. [23]

REST-arkkitehtuurityyli muodostuu kuudesta rajoitteesta: asiakas-palvelin-arkkitehtuurityyli, tilattomuus, välimuisti, kerroksellisuus, ladattava koodi ja yhtenäinen rajapinta. Asiakas-palvelin-arkkitehtuurityyli mahdollistaa asiakas- ja palvelinkomponenttien vastuiden erottamisen. Tämä kasvattaa asiakaskomponentin siirrettävyyttä eri alustojen välillä ja parantaa palvelimen skaalautuvuutta. Siirtämällä vastuu käyttöliittymästä asiakkaalle voidaan palvelimen toteutusta myös yksinkertaistaa. Lisäksi asiakas- ja palvelinkomponentteja voidaan kehittää toisistaan riippumatta. [23]

Toisen rajoitteen mukaan kaiken asiakkaan ja palvelimen välisen kommunikaation tulee olla tilatonta. Sovelluksen ja istunnon tila säilytetään kokonaisuudessaan asiakaskomponentissa. [23] Jokaisen asiakkaan tekemän palvelinpyynnön on oltava itsenäinen, ja pyyntö sisältää asiakkaan koko tilan. Näin ollen palvelimen ei tarvitse pyyntöön vastatakseen hakea sovelluksen tilaan liittyvää dataa muualta. [70] Järjestelmä palautuu paremmin osittaisista virheistä, mikä kasvattaa järjestelmän luotettavuutta. Myös skaalautuvuus paranee palvelimen resurssien hallinnan helpottuessa, koska palvelimen ei tarvitse huolehtia sovelluksen tilasta. [23]

Tilattomuus kuitenkin kasvattaa tarvittavan verkkoliikenteen määrää asiakkaan ja palvelimen välillä. Jokainen viesti sisältää sovelluksen tilan, joten viestit ovat kooltaan suurempia ja niitä on lähetettävä useammin. Tilan pysyessä samana muuttumatonta informaatiota lähetetään useita kertoja ja datan toisteisuus lisääntyy. [70] Lisäksi sovelluksen yhtenäinen ja oikeanlainen käyttäytyminen on riippuvaista erilaisten asiakasversioiden toteutuksen oikeellisuudesta, ja palvelimen kyky hallita sovelluksen yhtenäisyyttä heikkenee [23].

Välimuistia voidaan käyttää kompensoimaan esimerkiksi tilattomuuden aiheuttamia suorituskyvyn ongelmia. Välimuisti on toteutettavissa joko asiakaskomponenttiin, palvelimelle tai niiden välille esimerkiksi välityspalvelimelle. [70] Rajoitteen mukaan datan on sisällettävä tieto siitä, onko se mahdollista tallentaa välimuistiin. Tällaista voisi olla esimerkiksi data, jonka sisältämä tieto pysyy aina samana tai muuttuu vain harvoin. Asiakas kykenee hakemaan tallennetun datan suoraan välimuistissa, mikäli se vastaa aiemmin lähetettyä ja vastauksen saanutta palvelinpyyntöä. Välimuistin käyttö voi kuitenkin laskea järjestelmän luotettavuutta, jos asiakkaalle epähuomiossa tarjotaan vanhentunutta dataa, joka eroaa merkitävästi suoraan palvelimelta haetusta datasta. [23]

Neljäs rajoite lainaa kerrosarkkitehtuurityylin perusperiaatteita. Järjestelmä koostuu hierarkkisista kerroksista siten, että kukin kerros on tietoinen ainoastaan siitä kerroksesta, jonka palveluita se hyödyntää. [70] Tämä yksinkertaistaa järjestelmää ja edistää kerrosten itsenäisyyttä. Kerrosten avulla voidaan piilottaa vanhoja palveluita, suojata uudempia palveluita vanhoilta asiakastoteutuksilta ja yksinkertaistaa komponentteja esimerkiksi siirtämällä usein käytettyjä toiminnallisuuksia jaetulle välittäjäkomponentille. Välittäjäkomponentin käyttö mahdollistaa myös esimerkiksi kuormantasauksen, joka parantaa järjestelmän skaalautuvuutta. Kerrosten käytön heikkous on niiden aiheuttama viive dataa pro-

sessoitaessa, koska datan on kuljettava useiden kerrosten läpi. [23]

Viidennen rajoitteen, ladattavan koodin (engl. *code on demand*), toteuttaminen on aiemmin esitellyistä rajoitteista poiketen vapaaehtoista [70]. REST sallii asiakkaan toiminnallisuuksien laajentamisen tarjoamalla mahdollisuuden koodin lataamiseen palvelimelta esimerkiksi skripteinä. Tällöin valmiiksi toteutettujen ominaisuuksien määrää voidaan vähentää ja järjestelmän laajennettavuutta kasvattaa. Toisaalta järjestelmän läpinäkyvyys heikkenee. [23]

Yhtenäinen rajapinta (engl. *uniform interface*) määrittelee tavan käyttää yhteisiä resursseja. Resurssilla tarkoitetaan mitä tahansa asiaa, joka voidaan tunnistaa ja nimetä, jolle voidaan antaa osoite ja jota voidaan käsitellä internetissä. Resurssi voi olla esimerkiksi HTML-sivu, dokumentti, kuva, laite, palvelu tai resurssien kokoelma. [23, 70] Yhtenäisen rajapinnan saavuttamiseksi REST määrittelee neljä arkkitehtuurista rajoitetta, jotka ohjaavat komponenttien käytöstä. Nämä rajoitteet ovat resurssien käsittely esityksen kautta (engl. *manipulation of resources through representations*), itsekuvaavat viestit (engl. *self-describing messages*), resurssien tunnistaminen (engl. *identification of resources*) ja HATEOAS (engl. *hypermedia as the engine of application state*). [23]

REST-arkkitehtuurityylissä resursseja käsitellään resurssin tilaa kuvaavien esitysten (engl. *representation*) kautta. Kun asiakas pyytää palvelimelta tiettyä resurssia, palvelin vastaa asiakkaalle resurssin esityksellä, ei varsinaisella resurssilla. Esitys muodostuu joukosta tavuja sekä tavuja kuvaavasta metadatasta. [23] REST hyödyntää resurssien käsittelyssä HTTP-protokollan määrittelemiä metodeja, jotka ilmaisevat resurssille suoritettavan toimenpiteen. Tärkeimpiä näistä ovat POST, GET, PUT ja DELETE. Metodit vastaavat karkeasti perinteisiä CRUD-operaatioita create, read, update ja delete. [70] GET palauttaa resurssin esityksen, POST luo uuden resurssin, PUT päivittää olemassa olevan resurssin tilan ja DELETE poistaa resurssin [31].

Asiakkaan pyytäessä resurssia palvelimelta voidaan asiakkaan ja palvelimen välinen vuorovaikutus suorittaa ilman erillisiä neuvotteluja. Tähän viitataan itsekuvaavien viestien rajoitteella. Rajoitteen mukaan resursseilla on tietyt, valmiiksi sovitut esitysmuodot. Käytetyistä esitysmuodoista yleisimpiä ovat XML ja JSON. Käyttääkseen tiettyä resurssia asiakkaan on tunnistettava resurssi. REST määrittelee resurssin osoitteen sekä tavan resurssin käyttöönottoon ja tunnistamiseen URI-tunnisteen avulla. Tämä mahdollistaa sen, että resurssin tunniste voidaan muodostaa hyvin määritellyllä ja tunnetulla tavalla. HATEOS-rajoitteen mukaan asiakkaan vuorovaikutus palvelimen kanssa perustuu resurssien sisältämien linkkien käyttöön. Asiakas saa kaikki tarvitsemansa resurssit käyttöönsä seuraamalla linkkejä. Tällöin palveluiden käyttö ei vaadi erillistä vain tähän tarkoitukseen muodostettua tapaa palveluiden tunnistamiseen. [31]

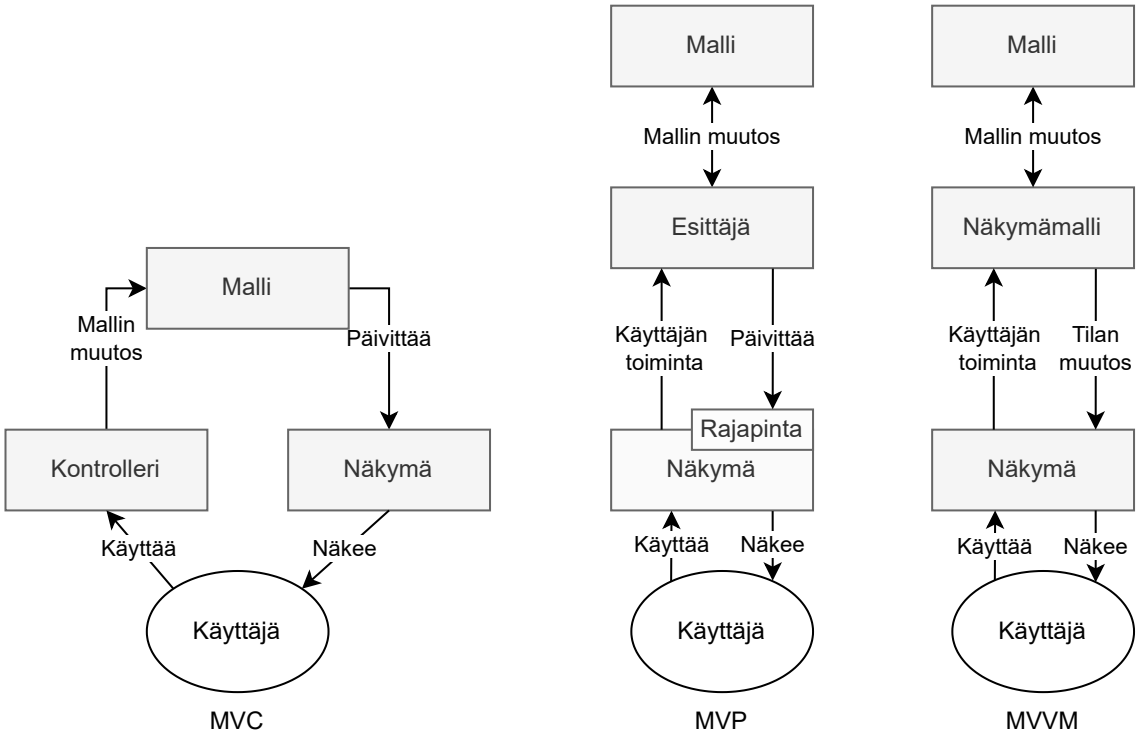
3.4 MV*-arkkitehtuurityylit

Termillä MV* (model-view-*) tarkoitetaan joukkoa arkkitehtuurityylejä, jotka vastuiden erottamisen periaatteen mukaisesti jakavat sovelluksen erillisiin komponentteihin. M viittaa malliin (model), V näkymään (view) ja * vaihtelevaan kolmanteen komponenttiin, joka auttaa hallitsemaan mallin ja näkymän välistä suhdetta. Päämääränä on helpottaa sovelluksen koodin suunnittelua, toteutusta ja ylläpitoa. MVC, MCP ja MVVM ovat tyyleistä tunnetuimpia, mutta MV*-arkkitehtuurityyliksi voidaan luokitella mikä tahansa vastaavaa komponenttijakoa toteuttava tyyli. [71] Tässä luvussa esitellään perusperiaatteet yleisimmistä MV*-arkkitehtuurityyleistä. Nämä on esitetty kuvassa 3.2. Tyylien yksityiskohdat voivat kuitenkin erota kuvassa esityistä toteutuksesta riippuen. Esimerkiksi MVC-tyylissä komponenttien väliset yhteydet voivat olla kaksisuuntaisia.

Vanhin ja tunnetuin MV*-tyyleistä on MVC (model-view-controller), joka sisältää mallin ja näkymän lisäksi kontrollerikomponentin (engl. *controller*). Kontrolleri reagoi käyttäjän suorittamaan toimintoihin ja ottaa vastaan käyttäjän antamat syötteet. Tämän perusteella kontrolleri viestii mallille sovelluksen tilan muutoksista. [71] Malli (engl. *model*) vastaa sovelluksen datasta ja on yhteydessä tietokantaan. Malli voi suorittaa myös datan käsittelyä ja validointia ennen datan tallentamista tai näyttämistä. [36] Kontrollerilta saadun tiedon pohjalta malli suorittaa tarvittavan sovelluslogiikan ja tallentaa tai hakee dataa tietokannasta. Näkymä päivittyy mallissa tapahtuneiden muutosten mukaisesti, ja vastaa sovelluksen tilan näyttämisestä käyttäjälle. [71]

MVP-tyylissä (model-view-presenter) malli pyritään selkeämmin erottamaan muista komponenteista. Kontrollerin sijaan käytetään esittäjäkomponenttia (engl. *presenter*), joka sijaitsee mallin ja näkymän välissä. Toisin kuin MVC-tyylissä, käyttäjän toimiin reagoi kontrollerin sijaan näkymä. Jokaista näkymää vastaa esittäjä, jolle näkymä välittää tiedon toiminnasta. Esittäjä sisältää sovelluksen esityslogiikan, ja se muodostaa yhdessä näkymän kanssa käyttöliittymän. Esittäjä on yhteydessä malliin datan tallentamiseksi ja hakemiseksi. Tilan muuttuessa esittäjä päivittää näkymää rajapinnan kautta. [71]

MVVM-tyylissä (model-view-viewmodel) mallin ja näkymän välissä sijaitsee näkymämalli (engl. *viewmodel*), joka pitää huolen siitä, että malli ja näkymä vastaavat toisiaan. Näkymämalli vastaa näkymän määrittelystä ja hallinnasta, ja sisältää esimerkiksi esityslogiikan ja datan muotoilun. [71] Näkymä ja näkymämalli on sidottu toisiinsa siten, että muutokset näkymämalliin päivittyvät automaattisesti näkymään [36].



Kuva 3.2. *Komponenttien välinen vuorovaikutus MVC-, MVP- ja MVVM-arkkitehtuurityyleissä.*

4. PERINTEISET JA MODERNIT WEB-SOVELLUKSET

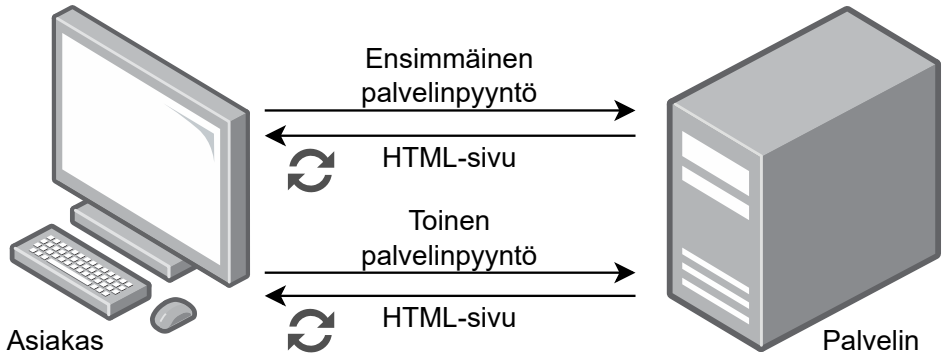
Web-sovellukset voidaan jakaa rakenteensa ja toimintaperiaatteidensa mukaan kahteen perustyyppiin, monen sivun sovelluksiin ja yhden sivun sovelluksiin. Monen sivun sovellukset edustavat perinteisempää käsitystä web-sovelluksista. Yhden sivun sovellukset puolestaan pyrkivät luomaan työpöytäsovelluksia muistuttavan käyttökokemuksen web-alustalle. Yhden sivun sovellusten alajoukon muodostavat RIA-sovellukset, joissa asiakaskomponentin ominaisuuksille asetetaan muita lisävaatimuksia osittaisen sivun päivityksen ohella.

4.1 Monen sivun sovellukset

Alun perin internet koostui yksinkertaisten tekstitiedostojen muodostamista dokumenteista. Dokumentit olivat rakenteeltaan staattisia ja niihin voitiin sisällyttää linkkejä muihin dokumentteihin. Rakenne soveltui hyvin tekstimuotoisen tiedon esittämiseen ja jakamiseen, mutta ei tarjonnut käyttäjälle muuta toiminnallisuutta. [34] Edellä kuvatun kaltaista toisiinsa linkitettyjen dokumenttien muodostamaa kokonaisuutta voidaan kutsua web-sivustoksi.

Pian kuitenkin havaittiin, että web tarjoaa varsin käyttökelpoisen alustan myös staattisten dokumenttien jakamista monimutkaisemmalle toiminnallisuudelle. Näin saivat alkunsa varsinaiset web-sovellukset, joissa asiakas-palvelin-arkkitehtuurityylin mukaisesti selain toimi käyttöliittymänä etäpalvelimella suoritettavalle ohjelmistolle. [34] Palvelin vastaa asiakkaan tekemiin pyyntöihin palauttamalla kokonaisia HTML-sivuja. Sivut ovat joko täysin staattisia tai ohjelmallisesti generoituja. [70] Palvelin voi esimerkiksi noutaa asiakkaan tarvitseman datan tietokannasta ja sisällyttää sen HTML-sivuun halutulla tavalla. Selaimen tehtäväksi jää ainoastaan palvelimen palauttaman sivun tulkinta ja näyttäminen käyttäjälle. [34]

Perinteiset web-sovellukset ovat rakenteeltaan monen sivun sovelluksia (engl. *multi-page application*) [34]. Kuten kuvassa 4.1 esitetään, palvelin vastaa jokaiseen asiakkaan tekemään pyyntöön kokonaisella HTML-sivulla. Tämän seurauksena koko käyttöliittymä on kerralla päivitettävä uuden sivun näyttämiseksi käyttäjälle [44]. Toiminta on synkronista, ja pyynnön lähetettyään selain jää odottamaan, kunnes palvelimen vastaus lopulta saapuu. [37]



Kuva 4.1. *Monen sivun sovelluksessa palvelin vastaa asiakkaan palvelinpyyntöihin kokonaisilla HTML-sivuilla, jolloin web-sovelluksen koko käyttöliittymä päivittyy.*

Web-sovellusten tarjoamien toiminallisuuksien lisääntyessä ja muuttuessa monimutkaisemmiksi asiakkaan täysi riippuvuus palvelimesta muodostui ongelmaksi. Esimerkiksi lomakkeen lähetettyään käyttäjän on odotettava datan siirtymistä verkon kautta palvelimelle, datan prosessointia, datan siirtymistä palvelimelta takaisin selaimelle ja lopulta koko näkymän päivittymistä [34]. Mikäli lomakkeen tiedoissa on virheitä, saa käyttäjä tiedon asiasta vasta uuden sivun päivittymisen jälkeen.

Jatkuva odottelu heikentää sovelluksen käytettävyyttä ja aiheuttaa turhautumista käyttäjissä [44]. Mahdolliset hitaat verkkoyhteydet ja monimutkainen käyttöliittymä voivat entisestään madaltaa käyttökokemusta. Tästä seuraa heikentynyt asiakastyytyväisyys, lyhyemmät vierailut verkkosivuilla, vierailujen määrän vähentyminen ja lopulta jopa liiketoiminnan tulojen pienentyminen. [72] Muunnettaessa työpöytäsovelluksia monen sivun web-sovelluksiksi onkin havaittu, että työpöytäsovelluksia vastaavaa käyttöliittymän interaktiivisuutta ja käyttökokemusta ei kyetä saavuttamaan [43]. Web-sovelluksen toteuttaminen monen sivun sovelluksena voi kuitenkin olla täysin perusteltua, mikäli sovellus muodostuu suurimmaksi osaksi staattisista dokumenteista, eikä tarkoituksena ole tarjota merkittävää määrää interaktiivista toiminnallisuutta.

4.2 Yhden sivun sovellukset

Toisin kuin monen sivun sovellukset, yhden sivun sovellukset (engl. *single-page application*) muodostuvat nimensä mukaisesti vain yhdestä HTML-sivusta, jonka osia päivitetään sovelluksen käytön aikana. Palvelin palauttaa HTML-sivun asiakkaan ensimmäisen palvelinpyynnön vastauksena. Tätä seuraaviin palvelinpyyntöihin palvelin vastaa vain pyydetyllä datalla, esimerkiksi JSON-muodossa. Myös HTML-sivun osien palauttaminen on mahdollista. Olennaista kuitenkin on, että palvelin ei ensimmäisen pyynnön jälkeen koskaan vastaa kokonaisella HTML-sivulla. [71]

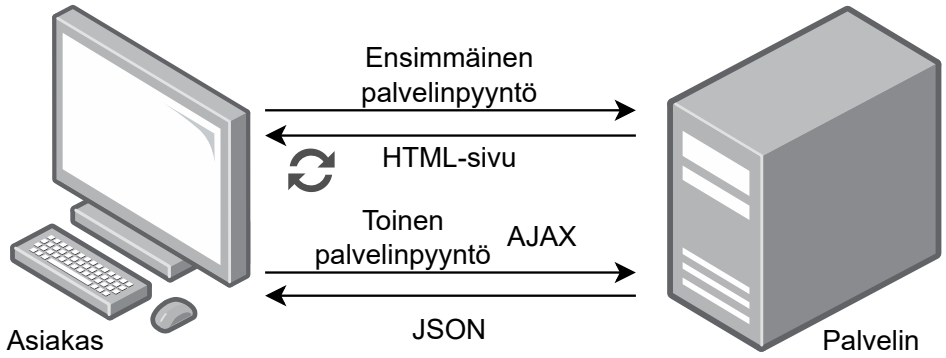
Yhden sivun sovelluksissa vastuu datan esittämisestä on usein täysin asiakkaalla, eikä

palvelimen tarvitse ottaa kantaa siihen, miten data muotoillaan. Asiakas yhdistää HTML:n ja palvelimelta vastaanottamansa datan halutulla tavalla. [71] Käyttöliittymä muodostuu erillisistä komponenteista, joita voidaan päivittää itsenäisesti [44]. Näin ollen vain sellaisia komponentteja, joiden data on muuttunut, päivitetään palvelinpyynnön seurauksena. Ainoastaan ensimmäisen palvelinpyynnön vastauksena saatava HTML-sivu aiheuttaa koko käyttöliittymän päivittymisen [71]. Yhden sivun sovelluksen palvelinpyyntöjä on esitetty kuvassa 4.2. Pyynnöt ovat asynkronisia, joten käyttöliittymä ei jäädy täysin ennen palvelimen vastauksen saapumista. Tämän mahdollistaa pyyntöjen toteuttaminen Ajaxin avulla. [44]

Ajax (asynchronous JavaScript and XML) on nimi useamman teknologian muodostamalla kokonaisuudelle ja web-sovellusten toteuttamisen lähestymistavalle. Ajax voidaan kuvata viiden periaatteen mukaan:

- Käyttöliittymä toteutetaan standardeihin perustuvilla teknologioilla, HTML-merkintäkielellä ja CSS-tyyleillä [29].
- Dynaaminen ja interaktiivinen käyttökokemus saadaan aikaan DOM-puuta muokkaamalla [29].
- Tiedonvaihtoon ja datan käsittelyyn käytetään XML-muotoa [29]. Nykyisin vaihtoehtoiseksi muodoksi soveltuu esimerkiksi JSON.
- `XMLHttpRequest` mahdollistaa asynkronisen kommunikaation määrittelemällä rajapinnan, joka tarjoaa toiminnallisuutta datan siirtämiseen asiakkaan ja palvelimen välillä [29, 94]. Modernimpana vaihtoehtona voidaan hyödyntää `Fetch`-rajapintaa [2].
- JavaScript yhdistää teknologiat yhtenäiseksi kokonaisuudeksi [29]. Koska Ajax perustuu JavaScriptiin, ei sen käyttö vaadi erikseen ladattavia teknologioita tai selaimen laajennuksia [51].

Ajaxia käytettäessä käyttöliittymän kokoamisesta ja kommunikaatiosta palvelimen kans-



Kuva 4.2. *Yhden sivun sovelluksessa vain ensimmäinen palvelinpyyntö aiheuttaa koko käyttöliittymän päivittymisen.*

sa vastaa JavaScriptilla toteutettu Ajax-moottori, joka hyödyntää Ajaxin keskeisiä teknologioita. Sellaiset käyttäjän toimet, jotka tavallisesti generoisivat palvelimelle lähetettävän HTTP-pyynnön, aiheuttavat JavaScript-kutsun moottorille. Moottori käsittelee itse tilanteet, joissa palvelimelle ei ole tarvetta, kuten yksinkertaisen datan validoinnin tai datan muokkaamisen muistissa. Mikäli moottorin on vastauksen muodostaakseen esimerkiksi lähetettävä tai haettava dataa palvelimelta, suoritetaan palvelinpyynnöt asynkronisesti niin, että käyttäjän vuorovaikutusta sovelluksen kanssa ei tarvitse estää. [29] Sovellus kykenee suorittamaan palvelimen kanssa käytävän kommunikaation taustalla ja päivittämään vain tiettyjä käyttöliittymän osia kerrallaan. Näin ollen käyttöliittymän päivitys ei vaadi koko näkymän uudelleen lataamista. [51]

Ajaxin ansiosta yhden sivun sovellukset mahdollistavat työpöytäsovelluksia muistuttavan käyttökokemuksen toteuttamisen web-ympäristössä. Palvelin vastaa asiakkaan pyyntöihin vain tarvittavalla datalla, mikä nopeuttaa ja keventää palvelimen ja asiakkaan välistä vuorovaikutusta. Koska palvelimen vastaukset muodostuvat datasta kokonaisten HTML-sivujen sijaan, ei sovelluksella ole tarvetta ladata ja päivittää koko käyttöliittymää kerralla. Tämä yhdessä asynkronisten pyyntöjen kanssa sujuvoittaa sovelluksen käyttöä ja vähentää tarvittavaa odottelua. [71] Yhden sivun sovelluksissa hyödynnetään asiakaslaitteen prosessointitehoja, ja selaimen vastuu sovelluksen suorituksesta kasvaa. Myös selaimen monipuolisten grafiikkamahdollisuuksien käyttöönotto on helpompaa. [72]

4.3 RIA-sovellukset

Termiä “Rich Internet Application”-käytettiin ensimmäisen kerran Macromedian raportissa vuonna 2002. Raportissa pohdittiin perinteisten työpöytä- ja web-sovellusten yhdistämistä sekä siihen liittyviä etuja ja haittoja. [1] RIA-sovellukset viittaavat hyvin monimuotoiseen joukkoon web-sovelluksia, jotka pyrkivät saavuttamaan työpöytäsovelluksia muistuttavan käyttökokemuksen [28]. RIA-sovellukset voidaan luokitella yhden sivun sovelluksiksi, mutta ne asettavat asiakaskomponentille ylimääräisiä vaatimuksia osittaisen sivun päivityksen lisäksi. Sovellusten toteuttamisessa voidaan hyödyntää JavaScriptin, CSS:n ja HTML:n ominaisuuksien lisäksi esimerkiksi selaimiin ladattavia lisäosia ja selainkohtaisia merkintäkieliä [16].

RIA-sovellukset tunnistetaan joukosta niille tyypillisiä ominaisuuksia, mutta jokaisen sovelluksen ei kuitenkaan tarvitse sisältää niitä kaikkia. [20] Usein pyritään monipuolisiin ja interaktiivisiin käyttöliittymiin sekä asiakkaan ja palvelimen välisen kommunikaation vähentämiseen. RIA-asiakas ladataan alussa alustavan datan kanssa, minkä jälkeen asiakaskomponentti vastaa datan esittämisestä ja tapahtumien käsittelystä. Palvelinpyyntöjä tehdään vain silloin, kun tarvitaan palvelimen tallentamaa informaatiota tai lähetetään dataa esimerkiksi lomakkeen lähetyksen yhteydessä. [16]

Perinteiset web-sovellukset säilyttävät datan ja sovelluksen tilan palvelimella, jolloin asiak-

kaan on tietoa tarvitessaan aina pyydettävä sitä erikseen palvelimelta. RIA-sovelluksissa dataa on mahdollista tallentaa myös asiakaskomponentin puolelle. [28] Tallennettava data on joko pysyvää tai sitä voidaan säilyttää vain tietyn ajan [20]. Esimerkiksi tieto ostoskorin sisällöstä on mahdollista säilyttää asiakkaalla ennen ostotapahtuman suorittamista ja datan lähetystä palvelimelle [28]. Asiakas voi lähettää kerralla suuremman määrän tallentamaansa dataa palvelimelle sopivalla hetkellä, mikä vähentää jatkuvan kommunikaation tarvetta. Kommunikaatio on asynkronista, joten asiakkaan ja palvelimen välinen vuorovaikutus tapahtuu taustalla siten, että koko käyttöliittymä ei jäädy operaation ajaksi. RIA-asiakas kykenee myös hakemaan dataa tarvittaessa useista eri lähteistä, kuten tietokannoista, palveluista ja rajapinnoista. Dataa voidaan käyttää monissa eri muodoissa, kuten raakadatana, XML- tai JSON-muodossa. [66]

Vanhemmissa web-sovelluksissa sovelluslogiikka suoritetaan lähes kokonaan palvelimella. RIA-sovelluksissa osa laskennasta voidaan siirtää asiakkaan vastuulle. Tämän ansiosta käyttöliittymä reagoi nopeammin käyttäjän toimiin, ja asiakkaan ja palvelimen välisen kommunikaation määrää voidaan vähentää. Käyttäjä kykenee esimerkiksi selaamaan, suodattamaan ja muokkaamaan ostoskorin tietoja ilman, että jokainen toiminto vaatii yhteydenottoa palvelimeen. [28] RIA-sovellukset saattavat sisältää offline-tilan, johon siirrytään, mikäli yhteyttä palvelimeen ei ole saatavilla [16]. Perinteisistä web-sovelluksista poiketen RIA-sovelluksissa asiakkaan ja palvelimen välisen kommunikaation voi aloittaa myös palvelin [28].

RIA-sovellukset ovat yhden sivun sovelluksia, joten kokonaisten sivujen hakemisen sijaan yksittäisiä käyttöliittymän osia on mahdollista päivittää. Tarpeen mukaan voidaan hakea myös skriptejä ja muuta esityslogiikkaa. [20] Käyttöliittymässä pyritään interaktiivisuuteen ja monipuolisiin kontrolleihin, ja sovelluksen on tuettava multimediasisältöä ja animaatioita. Lisäksi käyttöliittymän tulee olla käytettävissä useilla eri laitteilla ottaen huomioon myös mobiililaitteet. Asiakaskomponentti kykenee vaikuttamaan ajoympäristön oletuskäyttäytymiseen esimerkiksi muuttamalla selaimen takaisinpainikkeen toimintaa. [66]

5. WEB-TEKNOLOGIAT

Tyypillisesti web-sovellusten toteutuksessa hyödynnetään olemassa olevia kirjastoja, rajapintoja ja teknologioita. Tässä luvussa esitellään kolme web-sovellusten toteutusta tukevaa teknologiaa. Modernisointimenetelmistä kerätyn aineiston perusteella JSP osoittautui yleiseksi monen sivun sovelluksissa. Angular ja React puolestaan edustavat tällä hetkellä suosittuja yhden sivun sovelluksen teknologioita.

5.1 JSP

JSP (JavaServer Pages) on palvelinpuolen Java-teknologia, jolla voidaan toteuttaa monen sivun sovelluksia. JSP-sivut, servletit ja web-palveluiden päätepisteet (engl. *endpoint*) ovat Java-ohjelmointikielen web-komponentteja, joita käytetään laajentamaan web-palvelimen dynaamisia ominaisuuksia. Web-komponenttien toimintaa tukee web-kontti (engl. *web container*). Kontti on ajonaikainen alusta, joka tarjoaa erilaisia palveluita ja rajapintoja komponenteille. Palvelut voivat liittyä esimerkiksi palvelinpyyntöjen lähettämiseen, turvallisuuteen, rinnakkaisuuteen tai elinkaaren hallintaan. [84] Kontti myös määrittelee palvelinpyynnön käsittelevän komponentin pyynnön sisältämän URL-osoitteen mukaan. Osoite sisältää aliaksen, jonka perusteella kontti päättelee oikean komponentin. [75]

Asiakkaan lähettäessä HTTP-pyynnön palvelin muuntaa pyynnön `HTTPServletRequest`-olioksi. Olio välitetään web-komponenteille, jotka ovat vuorovaikutuksessa muiden luokkien tai tietokannan kanssa tuottaakseen dynaamista sisältöä. Komponentti voi itse luoda `HTTPServletResponse`-olion tai siirtää pyynnön toiselle komponentille. Lopulta palvelin muuntaa olion HTTP-vastaukseksi ja palauttaa sen asiakkaalle. [84] Usein JSP-teknologian yhteydessä käytetään MVC-arkkitehtuurityyliä toiminnallisuuden hajauttamiseksi ja olioiden välisten kytkentöjen vähentämiseksi. JSP-sivu vastaa näkymää, ja toteutuksesta riippuen erilaiset servlet- ja JavaBeans-luokat kontrolleria ja mallia. [81]

Servletit ovat Javan luokkia, jotka laajentavat web-palvelinten ominaisuuksia käsittelemällä palvelinpyyntöjä ja muodostamalla niihin vastauksia [84]. Servletille pyynnön tyypillä ei ole merkitystä, mutta yleensä niitä käytetään HTTP-pyyntöjen käsittelyyn web-palvelimilla. Tähän tarkoitukseen Java tarjoaa HTTP-protokollan käsittelyyn suunnattuja servlet-luokkia. [86] Servletit soveltuvat parhaiten palvelusuuntautuneisiin sovelluksiin, ja

esimerkiksi web-palveluiden päätepisteet toteutetaan servletteina. Palvelut, jotka vaativat pyyntöjen lähettämistä tai muun kuin tekstimuotoisen datan käsittelyä, sopivat hyvin servletteina toteutettaviksi. [84]

JSP-sivut ovat suositeltu tapa staattisen sisällön muodostamiseen. Ne soveltuvat hyvin tekstipohjaisten merkintäkielien, kuten HTML, SVG tai XML, generointiin, mutta sisältävät myös servlettien tarjoamat dynaamiset ominaisuudet. [80, 84] Rakenteeltaan JSP-sivu on tekstidokumentti, joka sisältää staattista dataa ja dynaamista sisältöä muodostavia JSP-elementtejä. Staattinen data voidaan ilmaista missä tahansa tekstipohjaisessa muodossa. JSP-elementit puolestaan esitetään joko standardisyntaksilla tai XML-merkintäkielellä. Jokainen tiedosto voi kuitenkin sisältää vain yhtä syntaksia. XML-syntaksilla toteutettua JSP-sivua kutsutaan JSP-dokumentiksi, ja sitä voidaan muokata käyttäen XML-dokumenttien työkaluja ja rajapintoja. [85]

JSP-sivulla staattisen datan esittämisen oletusformaatti on HTML. Käytettävää merkintäkieltä voidaan muuttaa JSP-sivuun merkittävien direktiivien avulla. Merkintäkielen ilmaisevalla direktiivillä määritetään myös sivulla käytettävä koodaus, kuten UTF-8. [77] Direktiiveillä voidaan myös hallita web-kontin vastuulla olevaa JSP-sivun käännöstä ja suoritusta sekä tuoda tagikirjastoja [82, 85]. JSP tarjoaa tagien standardikirjaston, joka sisältää monille JSP-sovelluksille yhteisiä ydintoiminnallisuuksia. Tällaisia ovat esimerkiksi iteraattoreihin, ehdolliseen toimintaan, XML-dokumenttien muokkaamiseen, tietokannan käyttöön ja usein käytettyihin funktioihin liittyvät tagit. [79] Standarditagien lisäksi voidaan määritellä omia custom-tageja, joihin liitetään tiettyjä usein toistuvia tehtäviä [83].

Dynaamista sisältöä voidaan muodostaa hyödyntämällä Javan olioiden ominaisuuksia. Implisiittiset oliot ovat web-kontin luomia, ja sisältävät informaatiota esimerkiksi palvelinpyyntöön, web-sivuun, istuntoon tai sovellukseen liittyen. Nämä oliot ovat suoraan käytettävissä ilman lisätoimenpiteitä. Sovelluskohtaiset oliot puolestaan on ensin luotava ja niiden ominaisuuksien arvot erikseen asetettava. Olioiden käytön tarkoituksena on siirtää sovelluslogiikka olioihin, jotta JSP-sivuilla voidaan keskittyä sisällön esitystapaan. Olioiden luomiseen ja käyttämiseen JSP-sivujen sisältä suositellaan JavaBeans-komponentteja, joita voidaan luoda Javan standarditagien avulla. [76] JavaBeans-komponentit ovat luokkia, jotka mahdollistavat toiminnallisuuksien vaivattoman uudelleen käytön ja kokoamisen sovelluksiksi. Mikä tahansa tiettyjä JavaBeans-suunnitteluperiaatteita noudattava luokka on JavaBeans-komponentti. [78]

5.2 React

React on käyttöliittymien toteuttamiseen tarkoitettu JavaScript-kirjasto [55]. Stack Overflown kehittäjäkyselyn mukaan React oli vuonna 2021 suosituin web-teknologia [73]. Usein Reactilla toteutetaan yhden sivun sovellusten käyttöliittymiä [56]. Tarvittaessa React taipuu kuitenkin useampiin käyttökohteisiin. Reactia voidaan esimerkiksi käyttää yhdessä

muiden kirjastojen ja ohjelmistokehysten kanssa. Tarjolla on useita erilaisia ympäristöjä ja työkaluja, kuten Next.js, joka mahdollistaa Reactin käytön palvelinpuolen renderöinnissä. Gatsbyn avulla puolestaan voidaan toteuttaa staattisia web-sivustoja, ja React Native-kirjastoa käyttäen mobiilisovelluksia. [55, 58] Koko käyttöliittymää ei ole välttämätöntä toteuttaa Reactilla. Sen sijaan uusia ominaisuuksia voidaan lisätä olemassa olevaan sovellukseen sen verran kuin on tarpeen. Tällöin vanhaa koodia ei tarvise toteuttaa kokonaan uudelleen. [55, 56]

Käyttöliittymän kuvaamiseen voidaan käyttää JSX-kieltä, joka on JavaScriptin syntaksilaajennus [59]. JSX tuottaa React-elementtejä, jotka määrittelevät käyttöliittymän käyttäjälle näkyvät osat [59, 60]. Rakenteeltaan elementit ovat muuttumattomia (engl. *immutable*) olioita. Oliot vastaavat käyttöliittymän tilaa tietyllä hetkellä. React vertaa elementtejä niiden edeltäjiin, ja päivittää DOM-puun rakennetta tämän mukaisesti. [60] Reactissa esittämislogiikkaa ei ole erotettu muusta käyttöliittymän toiminnallisuudesta, kuten tapahtumien käsittelystä tai tilanhallinnasta. Sen sijaan React hyödyntää vastuiden erottamisen periaatetta, jossa erilliset komponentit sisältävät sekä JSX-elementit että JavaScriptilla toteutetun toimintalogiikan. [59]

Komponentit auttavat jakamaan käyttöliittymään itsenäisiin ja uudelleen käytettäviin osiin, joiden avulla voidaan koota monimutkaisia käyttöliittymiä [55, 57]. Komponentit toteutetaan joko luokkina tai funktioina. Käsitteellisellä tasolla ne vastaavat JavaScriptin funktioita. Komponenteille voidaan antaa syötteenä propseiksi kutsuttuja arvoja, ja ne palauttavat React-elementtejä. [57] Jokainen komponentti toteuttaa `render`-metodin, joka vastaa elementtien renderöinnistä [55]. Komponentti voi viitata ulostulossaan (engl. *output*) muihin komponentteihin, jolloin palautettavat React-elementit rakentuvat useamman komponentin muodostamista elementeistä [57]. Tällöin syntyy hierarkkinen rakenne, jossa vanhempikomponentit välittävät propseja lapsikomponenteilleen.

Komponentti on tilaton tai tilallinen. Tilallisilla komponenteilla on oma kapseloitu tilansa, josta muut komponentit eivät ole tietoisia. [55, 56] Tila voidaan kuitenkin välittää lapsikomponentille propsien avulla. Tätä kutsutaan ylhäältä-alas- tai yksisuuntaiseksi tietovirraksi (engl. *data flow*). Tilan omistaa aina jokin tietty komponentti, ja tila vaikuttaa komponentin itsensä lisäksi vain hierarkiassa alempana oleviin komponentteihin. [56] Kun komponentin tila muuttuu, sen palauttamat elementit päivitetään kutsumalla uudelleen `render`-metodia [55]. Monimutkaisempaan sovelluksen tilanhallintaan voidaan hyödyntää ulkoisia kirjastoja. Alun perin tähän tarkoitukseen suositeltiin Flux-kirjastoa, mutta nykyään suosittuja ovat esimerkiksi Redux ja MobX. [18, 26, 45, 61]

5.3 Angular

Angular on kehitysalusta, joka sisältää ohjelmistokehyksen, kokoelman kirjastoja ja ohjelmistokehityksen työkaluja. Kirjastojen avulla sovelluksen ominaisuuksia voidaan laajen-

taa esimerkiksi reititykseen, lomakkeiden hallintaan sekä asiakkaan ja palvelimen väliseen kommunikaatioon liittyvillä toiminnallisuuksilla. Kehitystyökalut mahdollistavat koodin tuottamisen, koontiversion (engl. *build*) valmistamisen, testaamisen ja päivittämisen. [14] Tavoitteena ovat nopeus, skaalautuvuus ja ylläpidettävyys [3, 14]. Angular myös hyödyntää useita oliosuuntautuneen ohjelmistosuunnittelun suunnittelumalleja (engl. *design patterns*), kuten kuorrutinta (engl. *decorator*) ja riippuvuusinjektiota (engl. *dependency injection*) [14].

Tyypillisesti Angularilla toteutetaan yhden sivun sovellusten asiakaskomponentteja työpöytä-, mobiili- tai web-ympäristössä [3, 8]. Tarvittaessa Angular-sovellusta on mahdollista suorittaa myös palvelimella, jolloin suorituskykyä voidaan parantaa varsinkin heikkotehoisilla laitteilla [12]. Angular on toteutettu TypeScriptillä, joka muodostuu vahvasti tyypitetystä JavaScriptistä [14, 88]. Stack Overflown kyselyssä Angular oli kolmanneksi suosituin web-kehityksen teknologia vuonna 2021 [73]. Angularia edelsi AngularJS-teknologia, jonka tuki kuitenkin päättyi vuoden 2021 loppuun [87]. Tässä luvussa käsitellään vain uudempaan Angular-alustaa.

Angular-sovellus rakentuu moduuleista, jotka kokoavat yhteen tiettyyn aihealueeseen, työnkulkuun tai toiminnallisuuteen liittyviä komponentteja ja muita kooditiedostoja. Tyypillisesti sovellus koostuu useista tiettyihin ominaisuuksiin erikoistuneista moduuleista, mutta pienessä sovelluksessa yksikin moduuli voi riittää. [10] Moduuli ilmoittaa käännöskontekstin sen sisältämille komponenteille, ja muodostaa toiminnallisia kokonaisuuksia liittämällä komponentteihin moduulin vastuualueeseen liittyvää koodia, kuten palveluita. Moduulit voivat myös tuoda (engl. *import*) toiminnallisuutta muista moduuleista. Moduulirakenne helpottaa monimutkaisten sovellusten kehittämisen hallintaa, suunnittelua ja uudelleenkäyttöä. [8]

Jokaisessa Angular-sovelluksessa on oltava vähintään yksi komponentti, joka yhdistää komponenttihierarkian web-sivun DOM-puuhun [8]. Komponentti muodostuu sovelluslogiikasta ja datan käsittelystä vastaavasta luokasta, HTML-mallipohjasta (engl. *template*) ja mahdollisista tyyleistä, jotka määrittelevät mallipohjan elementtien ulkonäön [7]. Luokka sisältää `@Component`-kuorruttajan, joka määrittelee luokan komponentiksi. Angular määrittelee joukon kuorruttajia, jotka muokkaavat luokkia ja niiden toimintaa liittämällä luokkiin metadataa. Metadatan avulla järjestelmä päättelee luokkien tyypin ja toimintatavan. [8] `@Component`-kuorruttaja määrittelee komponenttiluokalle CSS-valitsimen, jonka avulla komponenttiin viitataan mallipohjassa [14]. Jokaisella komponentilla on oltava oma CSS-valitsin [4].

Metadata sisältää myös tarvittavan tiedon komponentin käyttämistä palveluista. Komponentti voi välittää osan tarvitsemistaan toiminnallisuuksista palveluille, jotka ovat useampien komponenttien käytettävissä. Palvelu on yleensä `@Injectable`-kuorruttimella määritelty luokka, jolla on hyvin rajattu käyttötarkoitus. Palvelu voi myös hyödyntää toiminnas-

saan muita palveluita. Tyypillisiä palveluiden toteuttamia toiminnallisuuksia ovat esimerkiksi datan hakeminen palvelimelta, käyttäjän syötteen validointi ja lokitietojen kirjoittaminen. [11] Palvelut tarjotaan komponentille riippuvuuksina riippuvuusinjektion avulla, jolloin komponentin ei itse tarvitse vastata palvelun alustamisesta [6].

Lisäksi metadata linkittää HTML-mallipohjan komponenttiin osoiteviitteellä tai suoraan komponentin sisäisenä koodina. Komponentti ja mallipohja muodostavat yhdessä näkymän. Näkymät muodostavat hierarkian, jonka ansiosta useammasta näkymästä rakentuvia käyttöliittymän osia voidaan muokata, näyttää tai piilottaa kokonaisuuksina. Usein hierarkia sisältää näkymiä monesta komponentista. [9] Komponentin mallipohja muodostuu HTML:stä ja Angularin omasta merkintäkielestä. Mallipohja määrittelee, miten siihen liittyvä näkymä renderöidään. [14, 8] Angular tukee kaksisuuntaista datan sidontaa (engl. *two-way data binding*), jonka seurauksena muutokset mallipohjassa näkyvät suoraan myös sovelluksen datassa [8]. Vastaavasti muutokset datassa aiheuttavat automaattisesti näkymän päivittymisen [5]. Kaksisuuntaista sidontaa käytetään myös vanhempi- ja lapsikomponenttien samanaikaiseen tapahtumien kuunteluun ja tiedonjakoon [13].

Mallipohjissa DOM-puun rakenteen dynaamiseen muokkaamiseen voidaan käyttää direktiivejä [14]. Direktiivit ovat luokkia, jotka sisältävät `@Directive`-kuorruttimen. Kuorrutin määrittelee esimerkiksi direktiivin valitsimen, jolla sitä merkitään mallipohjan HTML-merkintäkielessä. Direktiivit jaetaan rakenteellisiin ja attribuuttidirektiiveihin. Rakenteelliset direktiivit muuttavat näkymien rakennetta lisäämällä, poistamalla tai muokkaamalla elementtejä DOM-puussa. Attribuuttidirektiivit muuttavat olemassa olevien elementtien ulkonäköä tai toimintaa, ja näkyvät HTML:ssä samaan tapaan kuin tavalliset attribuutit. [9]

6. MODERNISOINNIN MENETELMÄT

Tässä diplomityössä modernisoinnilla tarkoitetaan sellaisia toimenpiteitä, joilla alkuperäinen sovellus muunnetaan kohdesovellukseksi muuttamatta merkittävästi sovelluksen toiminnallisuutta tai sovelluslogiikkaa. Lähdemateriaaleissa käytetty termistö vaihtelee, ja tällaista prosessia voidaan kutsua esimerkiksi modernisoinniksi, refaktoroinniksi, uudelleen suunnitteluksi (engl. *re-engineering*) tai migraatioksi. Yhteistä kaikille menetelmille on se, että uuden sovelluksen toteutusta ei aloiteta tyhjästä, vaan muunnosprosessin pohjana käytetään vähintään lähdesovellusta kuvaavaa mallia. Tässä luvussa esitellään ensin yleisellä tasolla joukko erilaisia menetelmiä vanhentuneiden monen sivun sovellusten modernisoimikseksi yhden sivun sovelluksiksi. Tämän jälkeen tarkastellaan yksityiskohtaisemmin kolmea menetelmää.

6.1 Yleiskatsaus

Mesbah ja Deursen [44] esittelevät menetelmän, jossa monen sivun sovellus muunnetaan yhden sivun sovellukseksi web-sivujen luokittelua sekä navigointipolkujen ja itsenäisesti päivitettävien käyttöliittymäkomponenttien tunnistamista hyödyntäen. Menetelmä käsittelee ainoastaan käyttöliittymään kohdistuvia muutoksia, eikä palvelinpuolen toiminnallisuuteen oteta kantaa. Modernisointi perustuu lähdesovelluksesta laadittujen mallien käsittelyyn ja muuntamiseen. Lisäksi tutkimuksessa esitellään beta-versio työkalusta, jonka avulla osa menetelmän sisältämistä vaiheista voidaan automatisoida. [44] Yksityiskohtaisemmin menetelmään perehdytään luvussa 6.2.1.

Myös Rossi et al. [68], Garrido, Rossi ja Distante [30] ja Urbieta et al. [89] syventyvät tutkimuksissaan käyttöliittymän mallintamiseen ja modernisointiin. Tutkimusartikkeleissaan he käsittelevät web-sovellusten malleihin kohdistuvaa refaktorointia, RIA-sovellusten käyttöliittymän suunnittelua ja web-sovellusten käyttöliittymien modernisointia RIA-sovelluksiksi. Apuna mallinnuksessa ja refaktoroinnissa käytetään abstrakteja datanäkymiä (engl. *abstract data views*, ADV), jotka mahdollistavat käyttöliittymän rakenteen ja käyttäytymisen kuvaamisen korkealla tasolla [68, 89]. Tarkoituksena on selvittää, miten käyttöliittymän modernisointi voidaan suorittaa systemaattisesti modernisointiin suunniteltujen RIA-refaktorointien avulla. RIA-refaktorointeja on kehitetty web-sovellusten suunnitteluun ja mallintamiseen tarkoitettujen suunnittelumallien pohjalta. Menetelmässä refaktorointi koh-

distetaan lähdesovelluksen navigointi- ja käyttöliittymämalleihin. Refaktorointeja suoritetaan tiettyihin sovelluksiin osiin sen sijaan, että modernisoitaisiin kerralla koko sovellus. [68]

Oman näkemyksensä käyttöliittymämallien modernisoinnista esittävät Linaje, Preciado ja Sanchez-Figueroa [39] RUX-menetelmässään. Menetelmä sisältää kolme vaihetta. Ensimmäisessä vaiheessa lähdesovelluksen mallista poimitaan sovelluksen datamalli ja sovelluslogiikka, joiden perusteella muodostetaan alustava versio abstraktista rajapinnasta. Abstrakti rajapinta on käyttöliittymäesitys, joka on yhteinen kaikille laitteille ja kehitysympäristöille. Abstrakti rajapinta muodostuu liittimistä, mediasta ja näkymistä. Liittimet yhdistävä mediaelementit datakerrokseen sovelluslogiikan kautta. Mediaelementit ovat atomisia informaatioelementtejä, jotka ovat riippumattomia asiakkaan renderöintiteknologiasta. Mediaa voivat olla esimerkiksi kuvat, tekstit, audio tai videot. Näkymä kuvaa joukkoa informaatiota, jonka käyttäjä näkee samalla kertaa. Mediaelementit ryhmitellään näkymien sisälle. [39]

Seuraavassa vaiheessa muodostetaan konkreettisen rajapinnan luonnos abstraktin rajapinnan pohjalta. Konkreettinen rajapinta on abstraktin rajapinnan toteuma (engl. *realization*) tietylle joukolle laitteita. Konkreettinen rajapinta jaetaan tila-, aika- ja vuorovaikutusesityksiin. Tilaesitys kuvaa miten komponentit asetetaan käyttöliitymään, minkä kokoisia ne ovat ja miltä ne näyttävät. Aikaesitys on vastuussa animaatioiden ja muiden aikaan liittyvien toimintojen määrittelystä. Vuorovaikutusesitys kuvaa, miten sovellus reagoi tapahtumiin, joita käyttäjän toimet aiheuttavat. Viimeisessä vaiheessa luodaan lopullinen rajapinta valitun RIA-teknologian perusteella. Lopullinen rajapinta liittyy tiettyyn laitteeseen tai laitteiden joukkoon ja tiettyyn kehitysteknologiaan. Rajapinnat muodostuvat RUX-menetelmän sisältämän komponenttikirjaston komponenteista. Lisäksi menetelmään sisältyy visuaalinen mallinnuskieli, ja rajapintaluonnosten automaattiseen muodostamiseen kehitetty RUX-työkalu. [39]

Mallipohjaisen menetelmän ovat kehittäneet myös Rodríguez-Echeverría et al. [67]. He esittelevät menetelmän, jossa web-sovelluksen WebML-malli voidaan muuntaa RIA-sovelluksen malliksi. Myöhemmin tutkimusryhmä on osana MIGRARIA-projektia julkaissut joukon vanhentenuiden web-sovellusten modernisointiin keskittyviä tutkimusartikkeleita [20, 63, 64, 65, 66]. Artikkelien kuvaamassa menetelmässä sovelluksen esityskerros sekä osia sovelluslogiikasta siirretään asiakaskomponenttiin, jolloin jokainen tilapäivitys ei välttämättä vaadi palvelinpyyntöä. Palvelinpuolen koodi pysyy pääasiassa koskemattomana lukuunottamatta uutta yhteyskerrosta, joka vastaa Ajax-pyyntöjen käsittelystä. Menetelmään palataan tarkemmin luvussa 6.2.2.

Wang et al. [90] huomioivat menetelmässään käyttöliittymän lisäksi myös palvelinpuolen koodin vaatimat muutokset. Web-sivusta muodostetaan mallipohja, joka sisältää vain sivun muuttuvat osat. Tähän sisältyy yhtenäisenä alueena koko sivun sisältöosa, joten

kerralla päivittyvää osuutta ei saada rajattua kovin tarkasti. Sivulle lisätään JavaScriptia, joka käyttäjän painaessa linkkiä suorittaa AJAX-pyynnön, ja huolehtii lisäksi palautettavan HTML-merkkijonon sijoittamisesta vanhan sisällön tilalle. JSP-teknologialla toteutettua palvelinkoodia muokataan siten, että AJAX-pyyntöjen käsittely ja oikeanmuotoisen HTML:n palauttaminen on mahdollista. Lisäksi esitellään työkalu, joka automatisoi palvelinpuolen koodin muuntamiseen. Työkalun toiminta rajoittuu kuitenkin melko yksinkertaisiin HTML-sivuihin ja Java-koodeihin. [90]

Oh et al. [50] pyrkivät menetelmässään ottamaan laajemmin huomioon sovelluksen toiminnallisuutta. Menetelmässä vanhentunut sovellus modernisoidaan siten, että kokonaiset sivut voidaan muodostaa asiakaskomponentissa palvelimen sijaan. Lisäksi kuvataan miten linkit ja lomakkeet voidaan muuntaa käyttämään Ajax-toiminnallisuutta sekä miten taaksepäin navigointi voidaan mahdollistaa. Muutokset kohdistuvat asiakaskomponentin lisäksi palvelinpuolen koodiin. Menetelmä rajoittuu mallipohjaisiin JSP-sovelluksiin. Menetelmään palataan yksityiskohtaisemmin luvussa 6.2.3.

Kokonaisen sovelluksen tai käyttöliittymän modernisoinnin sijaan voidaan keskittyä yksittäisiin sovelluksiin osiin. Esimerkiksi Chun ja Deanin [19] menetelmässä listoja sisältävät JSP-sivut muunnetaan käyttämään Ajaxia. Tällaiset sivut sisältävät usein linkkejä, joiden avulla siirrytään listan sisältämille sivuille. Muunnosprosessi on automatisoitu modernisoitavan koodin valintaa lukuunottamatta. [19] Ying ja Miller [95] puolestaan esittelevät lähdekoodin refaktorointiin perustuvan menetelmän JSP-sovelluksen lomakkeiden muuntamiseksi Ajax-lomakkeiksi. Tällöin lomakkeen lähettäminen ei aiheuta koko sivun päivittymistä. Lisäksi huomiota on kiinnitetty erityisesti lomakkeen validointiin, joka tulisi suorittaa sekä asiakkaan että palvelimen puolella. Validointi tulisi toteuttaa myös lomakkeen sisällön muuttuessa, ei ainoastaan lähetyksen yhteydessä. Menetelmä sisältää työkalun, joka avustaa ohjelmoijaa ja osin automatisoi muunnosprosessin. [95]

Toisaalta Ajax-teknologian käyttöönoton alkuaikoina selainten JavaScript-tuki ja -yhteensopivuus vaihteli suuresti, eikä JavaScriptille ollut tarjolla kehittyneitä kehitysympäristöjä. Ammattimaista ohjelmointia JavaScriptilla pidettiin haastavana, eikä kieli tarjonnut tukea esimerkiksi monille olio-suuntautuneen ohjelmoinnin konventioille. [53] Puderin [53] lähestymistavassa Javalla kirjoitettu Ajax-pohjainen web-sovellus käännetään tarvittavin osin JavaScriptiksi. Tämän mahdollistaa tutkimusartikkelissa esiteltävä XMLVM-kieli, jota käytetään välikielenä Javan ja JavaScriptin välillä [53] .

Tieteellisestä lähdeaineistosta ei löydetty yhtäkään tutkimusartikkelia, jossa monen sivun sovellus modernisoitaisiin käyttämään nykyaikaista web-kehityksen teknologiaa, kuten Reactia tai Angularia. Esimerkiksi Verhaeghe et al. [91] esittää kuitenkin menetelmän GWT-teknologiasta Angulariin siirtymiseksi. Kyseessä on muunnos vanhemmasta yhden sivun teknologiasta uudempaan. Samir, Stroulia ja Kamel [69] puolestaan käsittelevät Java-pohjaisen työpöytäsovelluksen muuntamista yhden sivun web-sovellukseksi. Lisäksi

Kaushalya ja Perera [35] esittelevät menetelmän vanhaan AngularJS-teknologiaan perustuvan sovelluksen modernisoimiseksi React-sovellukseksi. AngularJS-React-siirtymään perehdytään yksityiskohtaisemalla tasolla esimerkiksi Manirahon [40] ja Ravenscroftin [54] teksteissä, jotka kuvavat muunnosprosessia ja siinä huomioon otettavia seikkoja. Systemaattisia menetelmiä modernisoinnin suorittamiseen tekstit eivät kuitenkaan esitä.

Tieteellisten julkaisujen ulkopuolelta monen sivun sovelluksen modernisointia yhden sivun sovellukseksi käsittelevät esimerkiksi Neves [46] ja Lim ja Nguyen [38]. Nevesin [46] artikkelissa 15 vuotta vanha, paljon koodia sisältävä ja useilla eri teknologioilla toteutettu monen sivun sovellus modernisoidaan lisäämällä sovellukseen React-koodia. Prosessin aikana sovellukseen toteutetaan hiljalleen uusia ominaisuuksia ja vanhaa toiminnallisuutta refaktoroidaan lisäämällä React-koodia vanhojen sivujen sisälle. Sovelluksen tilan säilyttäminen ja sen osien välinen kommunikointi tapahtuu edelleen palvelimella. Palvelin myös lisää datan sivuihin kaikkein dynaamisimpia sivuja lukuunottamatta, joissa asiakaskomponentti hakee datan JSON-rajapinnan kautta. Palvelin renderöi ja palauttaa kokonaisia sivuja asiakkaan palvelinpyyntöihin. Sovelluksen linkkiä painettaessa palvelinpyynnöt kuitenkin suoritetaan Ajax-kutsuina ja kokonaisen sivun päivittymisen sijaan päivitetään vain sivun muuttuneet osat. Lopputuloksena syntyy ratkaisu, jossa vanhat sivut renderöidään palvelimella ja uudet React-sivut asiakaskomponentissa. [46]

Lim ja Nguyen [38] puolestaan kuvaavat JSP-mallipohjia käyttävän Spring MVC -sovelluksen modernisointia AngularJS-teknologialla. JSP-mallipohjat muunnetaan AngularJS-mallipohjiksi, jolloin käyttöliittymän kokoaminen siirtyy palvelimelta asiakkaan vastuulle. Palvelimen ja asiakkaan välille toteutetaan REST-rajapinta, jonka kautta data siirtyy JSON-muodossa. Asiakas lisää palvelimelta vastaanottamansa datan mallipohjiin. Palvelimen koodia muutetaan siten, että se kykenee vastaamaan asiakkaan tekemiin palvelinpyyntöihin halutulla tavalla. Datan validointi suoritetaan sekä asiakkaan että palvelimen puolella. Tekstissä huomautetaan, että asiakaspuolen JavaScript- ja JSON-tiedostoissa esitetty data ja sovelluslogiikka ovat käyttäjän nähtävissä selaimessa. Tästä syystä esimerkiksi sovelluslogiikka, jota ei haluta julkisesti paljastaa, on syytä suorittaa palvelimella. [38]

6.2 Menetelmien yksityiskohtaisempi tarkastelu

Seuraavaksi esitellään yksityiskohtaisemmin kolme edellisessä luvussa lyhyesti kuvattua menetelmää. Tarkoituksena on muodostaa tarkempi käsitys siitä, millaisia vaiheita menetelmät sisältävät ja miten ne eroavat toisistaan. Ensin käsitellään menetelmää, jossa lähdesovelluksen toimintaa analysoimalla muodostetaan käyttöliittymän modernisoitu malli. Toisena perehdytään useamman tutkimusartikkelin muodostamaan MIGRARIA-projektiin. Viimeisenä tarkastellaan JSP-teknologialla toteutetun MVC-sovelluksen muuntamista yhden sivun sovellukseksi.

6.2.1 Käyttöliittymän mallipohjainen modernisointi

Mesbah ja Deursen [44] esittävät modernisointimenetelmän, joka perustuu web-sivujen luokitteluun ja tämän pohjalta suoritettavaan navigointipolkujen ja käyttöliittymäkomponenttien tunnistamiseen. Navigointipolkujen ja käyttöliittymäkomponenttien avulla muodostetaan modernisoitu käyttöliittymän malli. Prosessi koostuu viidestä vaiheesta: sivujen hakeminen, navigointipolkujen muodostaminen, käyttöliittymäkomponenttien tunnistaminen, yhden sivun käyttöliittymän mallin määrittely ja kohdekäyttöliittymän mallin muunnos. [44]

Sivujen hakemiseksi muodostetaan sovelluksen ajonaikainen peilikopio, joka jäljittelee lähdesovelluksen sivurakennetta ja toiminnallisuutta. Sovellusta tarkastellaan loppukäyttäjän näkökulmasta, eikä palvelinpuolen toteutusta oteta huomioon. Kopion luomiseen käytetään staattisia ja dynaamisia analyysitekniikoita. Staattisten tekniikoiden avulla sivuilta havaitaan esimerkiksi muille sovelluksen sisäisille sivuille johtavia linkkejä. Dynaaminen analyysi puolestaan auttaa tunnistamaan sivuja, joiden näyttäminen vaatii tiettyjä toimintoja tai olosuhteita, kuten lomakkeen lähettämistä tai tietyn datan syöttämistä. [44]

Toisessa vaiheessa selvitetään lähdesovelluksen navigointi- ja rakennemalli. Tämä vaatii sovelluksen navigointipolkujen tunnistamista. Navigointipolku on reitti, joka muodostuu käyttäjän selatessa sovellusta ja seuratessa linkkejä sovelluksen sivulta toiselle. Navigointipolut esitetään puurakenteena, jossa jokainen puun solmu esittää yhtä web-sivua. Solmujen väliset kaaret kuvaavat linkkejä sivujen välillä. Navigointipolkujen tunnistamiseksi sellaiset sivut, jotka ovat yhteydessä toisiinsa linkin välityksellä ja jotka muistuttavat riittävissä määrin toisiaan, luokitellaan yhteiseen ryhmään. Sivujen ryhmittely aloitetaan juurisolmusta eli sovelluksen etusivulta. Navigointipolku etsitään ryhmittelemällä samankaltaisia sivuja jokaisella puun navigointitasolla. Lopuksi suoritetaan navigointipolun siistiminen ja yksinkertaistaminen yhdistämällä kukin ryhmä yhdeksi solmuksi. [44]

Ryhmien muodostamiseen käytetään artikkelin laatijoiden kehittämää skeemaan perustuvaa samankaltaisuutta (engl. *schema-based similarity*). Sivujen syntaksipuun vertailun sijaan käänteisen suunnittelun (engl. *reverse engineering*) avulla muodostetaan skeemat, joille lasketaan muokkausetäisyys (engl. *edit distance*). Tällöin esimerkiksi kahden rivin taulukon sisältävä sivu luokitellaan samaksi seitsemän rivin taulukon sisältävän sivun kanssa. [44]

Navigointimallin muodostamisen jälkeen tunnistetaan käyttöliittymäkomponenttien kandidaatit. Tunnistaminen suoritetaan vertailemalla web-sovelluksen sivuja toisiinsa navigointipolun mukaisesti. Mikäli toisiinsa linkittyvien sivujen A ja B käyttöliittymien ja HTML-koodin erot ovat hyvin pieniä, voidaan sama toiminnallinen lopputulos saavuttaa päivittämällä vain sivun A osaa kokonaisen sivun vaihtumisen sijaan. Tällöin siitä sivun B-osasta, joka erottaa sen sivusta A, muodostetaan oma käyttöliittymäkomponentti. [44]

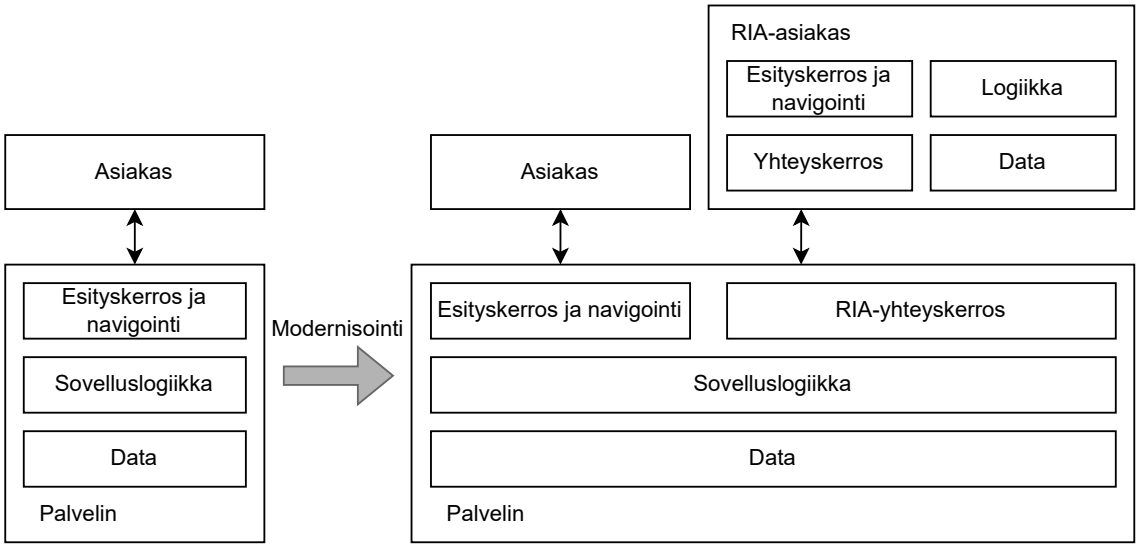
Aiemmissa vaiheissa muodostetun navigointimallin ja komponenttilistan avulla johdetaan käyttöliittymän metamalli. Tähän vaaditaan dynaamisten käyttöliittymien, navigointipolkujen ja yleisten Ajax-komponenttien mallinnukseen sopiva kieli. Artikkelin julkaisun aikaan tällaista ei suoraan ollut tarjolla, joten kirjoittajat olivat itse kehittäneet yhden sivun sovellukselle soveltuvaa käyttöliittymän metamallia. Metamalli koostuu widgeteistä, joista jokainen muodostuu joukosta käyttöliittymäkomponentteja. Komponentit ja niiden tarkempi rakenne määräytyvät käytettävän Ajax-kehyksen mukaan. Palvelimen widgetit ja komponentit generoivat näkymiä, jotka muodostavat asiakaskomponentin. Navigointi sovelluksessa perustuu näkymien tilan muuttumiseen. Viimeisessä vaiheessa käyttöliittymän metamalli muunnetaan kohdejärjestelmän teknologioita vastaavaan malliin, joka kuvataan alustaspesifillä kielellä. Tarvittaessa käyttöliittymän abstrakti metamalli voidaan muuntaa useamman eri Ajax-kehyksen tai toteutustavan mukaisesti. [44]

Modernisointimenetelmän lisäksi Mesbah ja Deursen [44] esittelevät artikkelissaan Retjax-prototyyppityökalun, jonka avulla voidaan suorittaa navigointipolun ja käyttöliittymäkomponenttien tunnistaminen. Työkalun toimivuutta testataan ja arvioidaan pienimuotoisen tapaustutkimuksen avulla. Tutkimuksessa havaitaan, että esimerkiksi huonosti muodostetut tai virheelliset HTML-elementit aiheuttavat työkalulle ongelmia. Lisäksi todetaan, että työkalu ei sovellu sellaisten web-sovellusten modernisointiin, jotka sisältävät pitkiä sisältöjä, kuten uutisia tai artikkeleita. [44]

6.2.2 MIGRARIA-projekti

MIGRARIA-projektin tavoitteena on määritellä systemaattinen ja osittain automatisoitu menetelmä vanhentuneiden web-sovellusten modernisoimiseksi RIA-sovelluksiksi. Menetelmässä hyödynnetään ADM-periaatteita (Architecture-Driven Modernization) ja MDD-tekniikoita (Model-Driven Development). [66] Lähdesovelluksen esityskerroksen ja navigointitiedon pohjalta muodostetaan joukon RIA-ominaisuuksia täyttävä asiakaskomponentti. Kuten kuvassa 6.1 esitetään, siirretään esityskerros ja navigoinnista vastaava toiminnallisuus asiakaskomponenttiin. Esityskerros on muunnettava vastaamaan yhden sivun sovelluksen rakennetta ja toiminnallisuutta. [66]

Navigointi sovelluksessa aiheuttaa käyttöliittymän tilan muutoksen. Tilan muutos määritellään päivitykseksi näkyvissä olevaan käyttöliittymäkomponenttiin tai uuden komponentin lataukseksi. Asiakaskomponentti voi tallentaa dataa ja suorittaa osan sovelluslogiikasta, jolloin jokainen tilapäivitys ei välttämättä vaadi palvelinpyyntöä. Asynkronisesta kommunikoinnista palvelimen kanssa vastaa asiakkaan yhteyskerros. Suurin osa palvelinkoodista pysyy ennallaan, joten myös RIA-ominaisuuksia tukematon asiakas voi edelleen käyttää palvelua. Palvelimelle kuitenkin lisätään uusi yhteyskerros RIA-asiakkaan pyyntöjen käsittelyyn alkuperäisen sovelluskerroksen yläpuolelle. Tässä otetaan huomioon myös jaettu sovelluslogiikka ja data asiakkaan ja palvelimen välillä. [66] Kerros toteutetaan REST-



Kuva 6.1. *Sovelluksen rakenteen muutos modernidisointiprosessissa.*

arkkitehtuurityylillä, ja se tukee JSON- ja XML-muotoista dataa. Yhteyskerros pyritään muodostamaan automaattisesti lähdesovelluksen pohjalta tunnistamalla resurssit ja generoimalla niille URI-tunnisteet. [65]

Modernisointiprosessi sisältää neljä vaihetta: informaation poimiminen lähdesovelluksesta, tiedon jalostus (engl. *refinement*) ja sisällyttäminen korkeamman abstraktiotason malliin, vapaaehtoinen mallin muuntaminen haluttuun muotoon ja tietylle konkretiatasolle, sekä lopullinen koodin generointi [66]. Rodríguez-Echeverría et al. [63] esittävät edelliseen perustuen hieman tarkennetun prosessin, jossa keskitytään MVC-pohjaisen web-sovelluksen mallintamiseen MDWE-tekniikoilla (Model Driven Web-engineering). Selkeyden vuoksi tässä luvussa käsitellään kuitenkin vain ensin mainittua nelivaiheista menetelmää.

Tavoitteena on muodostaa sovelluksesta malli, jonka avulla modernisointiprosessia voidaan yksinkertaistaa vähentämällä riippuvuutta lähdesovelluksen toteutusteknologiasta. Esimerkiksi palvelimen lähdekoodeille, tietokantojen skeemoille ja konfiguraatiotiedostoille suoritetaan staattinen analyysi. [64] Lopuksi sovellus analysoidaan dynaamisesti tutkimalla järjestelmän ajonaikaisia ominaisuuksia ja käyttäjän vuorovaikutusta sovelluksen kanssa [66].

Lisäksi selvitetään, mitkä vanhentuneen järjestelmän informaation osat ovat olennaisia modernisoinnin kannalta. Käyttöliittymän rakenteen tuntemus auttaa saavuttamaan lopputuloksen, joka näyttää ja tuntuu samalta kuin alkuperäinen sovellus. Jotta komponentit voidaan tunnistaa, tarvitaan tietoa web-sivujen ja niiden sisältämän datan suhteesta. Dynaamisesti sovelluksen suorituksen aikana generoitavat sivut voivat olla rakenteeltaan samanlaisia, mutta sisältää eri dataa. Tärkeää on selvittää myös sovelluksen navigointirakenne esimerkiksi navigointikartan muodossa. Navigointikarttaan sisältyy operaatio-

kartta, joka auttaa hahmottamaan palvelinpyyntöjen käsittelyä ja niiden aiheuttamia operaatioita palvelimella. [66] Lisäksi voidaan tunnistaa sivujen väliset linkit sekä selvittää, milloin ja millaisissa tilanteissa asiakas tekee palvelinpyyntöjä [64].

Edellisessä vaiheessa kerätyn tiedon pohjalta luodaan teknologiariippumaton tietomalli (engl. *knowledge model*), joka sisältää staattisen ja dynaamisen analyysin avulla tunnistetun informaation järjestelmästä. Tietomallista etsitään RIA-mallien (engl. *RIA-patterns*) ilmentymiä, ja havaitut ilmentymät merkitään malliin. [20] Jäljitettävyysmatriisien (engl. *traceability matrices*) avulla modernisointivaatimukset, RIA-ominaisuudet ja RIA-mallit voidaan linkittää toisiinsa. Tarkoituksena on selvittää, mikä RIA-ominaisuus vastaa mitäkin vaatimusta ja millä RIA-mallilla se voidaan toteuttaa. Lopuksi alkuperäisen tietomallin pohjalta luodaan uusi malli, joka sisältää valitut RIA-mallit ja -ominaisuudet. [20, 67] Muodostettua mallia voidaan halutessa tarkentaa konkreettisemalle tasolle ja käsitellä esimerkiksi MDWE-tekniikoilla. Näin voidaan helpottaa viimeisessä vaiheessa suoritettavaa koodin generointia. [66]

6.2.3 MVC-sovelluksen modernisointi

Oh et al. [50] ehdottavat menetelmää mallipohjiin perustuvan web-sovelluksen muuntamiseksi yhden sivun sovellukseksi. JSP-teknologialla toteutetun MVC-sovelluksen rakennetta uudistetaan siten, että web-sivut kootaan lopulliseen muotoonsa asiakaskomponentissa palvelimen sijaan. Tavoitteena on vähentää web-sivujen jakaman koodin ja datan turhaa muokkausta ja siirtoa. [49] Menetelmä noudattaa web-standardeja eikä vaadi ylimääräisiä kirjastoja tai laajennuksia. Lisäksi artikkelissa esitellään Eclipse-laajennus, jolla muunnosprosessi voidaan suorittaa automaattisesti. [50] Menetelmää esitellään kahdessa artikkelissa, joista jälkimmäisessä menetelmää tarkennetaan ja laajennetaan, sekä esitellään menetelmän toteuttava algoritmi.

Mallipohjia hyödyntävässä web-sovelluksessa mallipohjaa käytetään web-sivujen yhteisenä rakenteena ja pohjana, joka ylläpitää sivujen yhteisiä sisältöjä yhdessä paikassa [49]. Tyypillisesti mallipohja muodostuu HTML-tiedostosta, joka sisältää tiettyjä dynaamisesti generoitavia osia. Sivukohtainen sisältö yhdistetään mallipohjan kanssa tarpeen mukaan. [49, 50] Mallipohjan etuja ovat toisteisen koodin vähentyminen sekä sovelluksen ylläpidettävyyden ja uudelleen käytön helpottuminen. Perinteiset mallipohjaan perustuvat web-sovellukset ovat kuitenkin rakenteeltaan monen sivun sovelluksia, jolloin jokainen palvelinpyyntö aiheuttaa kokonaisen HTML-sivun lataamisen. [50]

Asiakkaan lähettäessä palvelinpyynnön palvelimen kontrolleri vastaanottaa pyynnön ja suorittaa pyyntöön liittyvän sovelluslogiikan. Lopputulos sisällytetään mallipohjaan, jolloin syntyy asiakkaalle palautettava sivu. Sivun tarvitsema data haetaan mallilta. Palvelin kokoaa mallipohjan ja sivukohtaisen sisällön yhteen ja lähettää sen asiakaskomponentille jokaisen palvelinpyynnön yhteydessä. [49] Ratkaisuna tähän menetelmä pyrkii vähentä-

mään asiakas- ja palvelinkomponentin välisen vuorovaikutuksen määrää välttämällä saman mallipohjan palauttamista asiakkaalle useampaan kertaan. Lisäksi estetään kokonaisten sivujen päivittyminen esimerkiksi lomakkeen lähetyksen jälkeen. [50]

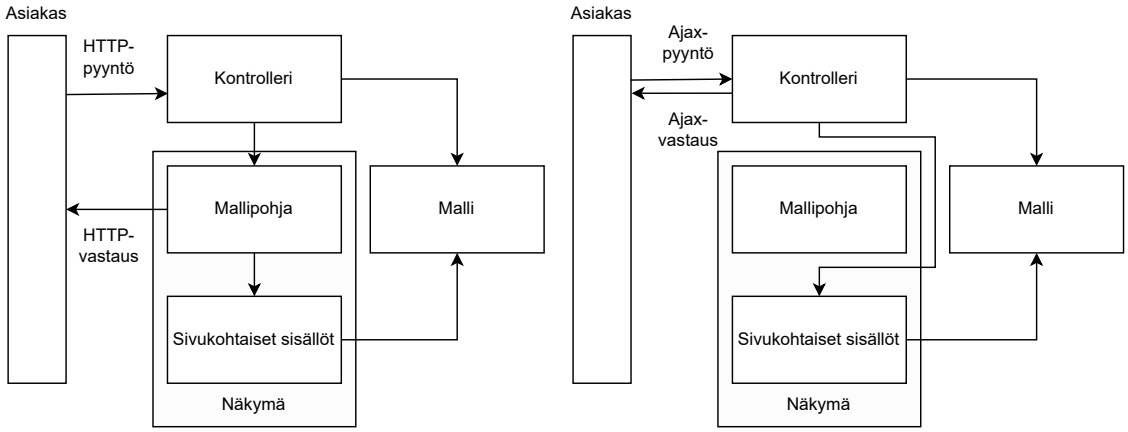
Uudistuksista huolimatta sovelluksen käyttöliittymän rakenne, sivujen navigaatiomalli, datamalli ja toiminnallisuus eivät muutu. Muutokset näkyvät parantuneessa vasteajassa ja siirrettävän datan määrässä. Asiakkaan ja palvelimen välinen vuorovaikutus muuttuu asynkroniseksi. Mallipohja ladataan vain ensimmäisen palvelinpyynnön yhteydessä, minkä jälkeen se yritetään mahdollisuuksien mukaan ohittaa myöhemmissä pyynnöissä. Asiakkaan pyynnöt muunnetaan Ajax-pyynnöiksi, jotka mahdollistavat sivujen osittaisen päivittämisen selaimessa. Kokonaiset sivut muodostetaan asiakaskomponentissa DOM-puuta päivittävän JavaScript-koodin avulla. [49]

Aluksi lähdesovelluksesta tunnistetaan dynaamisin ja staattisin analyysein sellaiset linkit ja lomakkeet, jotka aiheuttavat palvelinpyyntöjen lähettämisen palvelimelle. Jokaisesta pyynnöstä selvitetään sen dynaaminen arkkitehtuuri, joka voidaan esittää web-komponenttien välisiä suhteita kuvaavalla yhteistyökaaviolla (engl. *collaboration diagram*). Tämän jälkeen suoritetaan modernisointiin liittyvät muutokset toteuttava algoritmi. Kohdesovelluksesta laaditaan yhteistyökaavio, jota verrataan lähdesovelluksen kaavioon. Samalla arvioidaan muunnoksen onnistumista, ja virheitä havaittaessa voidaan tarvittavat muutokset suorittaa manuaalisesti. [49]

Menetelmä sisältää neljä vaihetta: muuttuvien osien tunnistaminen mallipohjasta, linkkien muuntaminen, lomakkeiden muuntaminen sekä taaksepäin navigoinnin ja kirjanmerkkien käsittely. [50] Muunnoksia suoritettaessa on selvitettävä, onko tietty palvelinpyyntö mahdollista muuntaa sivukohtaisen sisällön pyynnöksi. Esimerkiksi mikäli pyynnön muuntaminen aiheuttaa sovelluksen toiminnallisuuteen muutoksia, ei muuntamista suoriteta. Kokonaisen yhden sivun sovelluksen sijaan lopputulos voikin olla myös monen sivun sovellus, jossa yksittäiset sivut toimivat yhden sivun sovelluksina. [49]

Ensimmäisessä vaiheessa mallipohjasta tunnistetaan dynaamisesti generoitavat sivukohtaiset osiot, jotka voidaan muuntaa itsenäisesti päivittyviksi elementeiksi. Näihin liitetään esimerkiksi uniikeilla `id`-attribuuteilla merkityt `div`-elementit, jotka esittävät erillisiä osioita HTML-dokumentissa. Tällöin voidaan rakentaa käyttöliittymä, joka mahdollistaa yksittäisten `div`-osioiden päivittämisen käyttäjän selatessa sovellusta. Toisessa vaiheessa sovelluksen sisäiset linkit refaktoroidaan siten, että niiden painaminen ei aiheuta kokonaisen sivun vaihtumista. Linkkeihin lisätään linkin painalluksen käsittelevä tapahtumankäsittelijä, ja linkin osoite siirretään tapahtumakäsittelijän parametriksi. Tapahtumankäsittelijä lähettää Ajax-pyynnön, johon palvelin vastaa esimerkiksi HTML-elementillä. Vastauksena saatu HTML lisätään nykyiseen sivuun. Linkit luokitellaan kahteen kategoriaan sen mukaan, onko mallipohjan osoitetta käytetty linkin URL-osoitteessa. Linkki voi viitata mallipohjaan joko suoraan tai epäsuorasti. [50]

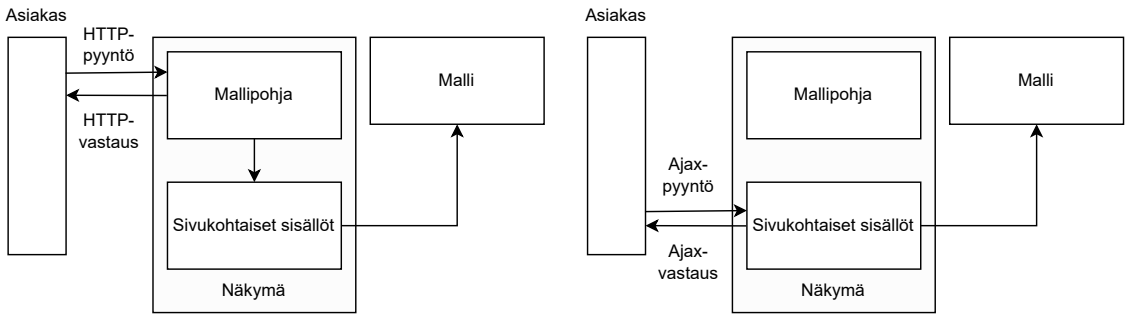
Epäsuorasti malliin viittaava linkki ei sisällä mallin osoitetta. Tällaista linkkiä painettaessa asiakaskomponentin lähettämän palvelinpyynnön vastaanottaa kontrolleri. Kontrolleri suorittaa tarvittavan sovelluslogiikan ja välittää pyynnön mallipohjalle, joka generoi sivun. Modernisoinnissa järjestelmää muutetaan siten, että kontrolleri kutsuu näkymän sivukohtaista sisältöä suoraan mallipohjan ohi. Tällöin kokonaisen HTML-sivun palauttaminen palvelimelta voidaan estää. [50] Tämä esitetään kuvassa 6.2. Suoraan mallipohjaan viittaava linkki sisältää mallipohjan käyttämän sivukohtaisen sisällön osoitteen. Tällöin päivitettävä sivun osa on staattinen HTML-tiedosto, jonka osoite on tiedossa, ja palvelinpyyntö voidaan tehdä suoraan mallipohjalle. Mallipohja muodostaa kokonaisen sivun, johon linkissä viitattu HTML yhdistetään. Kuten epäsuorien linkkien tapauksessa, halutaan mallipohja ohittaa kokonaan. Modernisoidussa linkissä osoitetietoa käytetään suoraan sivukohtaisen sisällön kutsumiseen kuvan 6.3 mukaisesti. [50]



Kuva 6.2. *Vasemmalla HTTP-pyynnön kulku epäsuorasti mallipohjaan viittaavan linkin tapauksessa ennen modernisointia. Oikealla Ajax-pyynnön kulku samassa tilanteessa modernisoinnin jälkeen, jolloin mallipohja ohitetaan.*

Linkkien lisäksi modernisoidaan lomakkeet, jotta lomakkeen lähetys ei aiheuttaisi koko sivun päivittymistä. Lomake-elementin attribuutteihin tehdään muutoksia ja lomakkeen lähetystapahtumaan asetetaan Ajax-funktio. Näin estetään lomakkeen oletuskäyttäytyminen ja lähetyspainikkeen toiminnallisuus voidaan määrittää uudelleen. Kun lomake lähetetään, funktio hakee datan lomakkeesta ja lähettää sen Ajax-pyynnön mukana palvelimelle. GET- ja POST-pyynnöt lähettävät datan eri muodossa, joten niille toteutetaan omat funktionsa. Pyyntöihin lisätään URL-osoite, jolloin lomake käyttäytyy samalla tavoin kuin modernisoidut linkit. [50]

Linkin painaminen muuttaa sovelluksen tilaa, mutta ei vaikuta selaimen URL-osoitteeseen. Mikäli käyttäjä yrittää palata yhden sivun sovelluksessa edelliselle sivulle, lataa selain edellisen URL-osoitteen mukaisen web-sivun, jolla käyttäjä vieraili ennen sovelluksen avaamista. Näin ollen sovelluksen tilasta toiseen ei voida liikkua käyttäen selaimen edellinen- ja seuraavapainikkeita. Tämä voidaan ratkaista käyttämällä HTLM5:n `history`-rajapintaa. Rajapinnan funktioita kutsumalla selaushistoriaan voidaan lisätä so-



Kuva 6.3. *Vasemmalla HTTP-pyynnön kulku suoraan mallipohjaan viittaavan linkin tapauksessa ennen modernisointia. Oikealla Ajax-pyynnön kulku samassa tilanteessa modernisoinnin jälkeen, jolloin mallipohja ohitetaan.*

velluksen tiloja siten, että tilojen välillä liikkuminen on mahdollista. `History`-rajapintaa hyödyntämällä voidaan sovelluksen tilaa vastaava URL saada näkyviin myös selaimen osoitekenttään, jolloin sovelluksen tila on mahdollista lisätä kirjanmerkiksi. Tilan URL-osoite muodostetaan tilanmuutokseen liittyvän Ajax-kutsun uniikin osoitteen perusteella. [50]

7. POHDINTA

Tässä luvussa esitellään työn toteutuksen aikana suoritettua tiedonhankintaa ja vastataan asetettuihin tutkimuskysymyksiin. Aluksi käsitellään lähdeaineiston hakua ja valintaa, sekä pohditaan siinä ilmenneitä haasteita. Tämän jälkeen esitellään modernisointimenetelmien analysoinnin ja muun soveltuvan aineiston pohjalta saavutet tulokset ja johtopäätökset.

7.1 Tiedonhaku

Työ toteutettiin kirjallisuusselvityksenä. Aineistona käytettiin erilaisia verkkomateriaaleja, teknisiä dokumentaatioita, kirjoja ja tieteellisiä artikkeleita. Erityisesti modernisointimenetelmiin liittyvää materiaalia etsittäessä pyrittiin painottamaan tieteellisiä lähteitä. Tieteellisiä artikkeleita ja tutkimuksia haettiin ACM Digital Library-, Google Shcolar-, IEEE Electronic Library- ja Scopus-tietokannoista ja –hakukoneista. Termistön moninaisuus aiheutti haasteita hakusanojen valinnassa. Työssä käsitellyn kaltaista modernisointiprosessia voidaan kuvata esimerkiksi termeillä “modernization”, “refactoring”, “re-engineering” tai “migration”. Yhden sivun sovelluksiin viittaavia käsitteistä ovat tilanteesta riippuen esimerkiksi “spa”, “single-page application”, “ajax”, “ria”, “rich internet application” ja “web 2.0”. Lisäksi aineistoa haettiin suoraan modernisoinnnin lähde- ja kohdeteknologioiden nimillä, kuten "JSP", "React"tai "Angular". Hakuja suoritettiin käyttäen termejä joko yksin tai useampia termejä loogisilla operaattoreilla yhdistäen.

Menetelmien toimivuudesta esitetyt numeeriset arvot eivät ole suoraan vertailukelpoisia, ja hyvin monissa tutkimuksissa niitä ei esitetä lainkaan. Näin ollen vastauksia tutkimuskysymyksiin on etsitty laadullisella analyysilla. Yhteisiä johtopäätöksiä on pyritty muodostamaan useamman menetelmän perusteella induktiivisen päättelyn avulla. Modernisointimenetelmiin tavoitteena oli valita sellaisia tutkimuksia, joissa esitetään toistettava ja systemaattinen menetelmä monen sivun sovelluksen modernisoinnista yhden sivun sovellukseksi. Täten esimerkiksi modernisointi työpöytäsovelluksesta tai toisesta yhden sivun sovelluksen teknologiasta on rajattu tarkastelun ulkopuolelle, vaikkakin näistä lyhyesti menetelmien esittelyssä mainitaan.

Tieteellisen lähdeaineiston kokoaminen osoittautui oletettua haasteellisemmaksi. Tuoretta tieteellistä tutkimusta, joka vastaisi täysin työssä käsiteltyä aihetta, ei onnistuttu lain-

kaan löytämään. Uusin tieteellinen artikkeli on vuodelta 2016, joten uudemmat web-teknologiat, kuten React ja Angular eivät esiinny analyysiin käytetyssä tieteellisessä aineistossa. Tästä syystä mukaan valittiin myös muutamia aihetta käsitteleviä artikkeleita tieteellisten aineistojen ulkopuolelta. Monien ei-tieteellisten artikkelien ongelmaksi muodostui kuitenkin se, että ne eivät esitelleet kokonaisia modernisoinnin menetelmiä. Sen sijaan teksteissä keskityttiin jakamaan yleisiä vinkkejä ja vihjeitä modernisointiprosessin tueksi.

7.2 Tulokset ja johtopäätökset

Vanhentuneiden monen sivun sovellusten modernisoinnin menetelmissä pyritään tyypillisesti systemaattisiin ja toistettaviin prosesseihin, jotka vähentävät modernisoinnin virhealttiutta ja sen aiheuttamaa työmäärää. Monissa menetelmissä alkuperäisestä sovelluksesta tai sen käyttöliittymästä muodostetaan malli, jonka perusteella sovellus muunnetaan yhden sivun sovellukseksi [39, 44, 48, 68]. Mallit mahdollistavat toteutusteknologiasta riippumattoman modernisoinnin.

Usein menetelmien yhteydessä esitellään työkalu tai algoritmi, jonka avulla mallien muuntaminen voidaan automatisoida [44, 68]. Myös mahdollisia koodimuutoksia suorittavat tai mallin pohjalta automaattisesti koodia generoivat työkalut ovat yleisiä [50, 90]. Tyypillisesti työkalut sisältävät kuitenkin rajoitteita koodin monimutkaisuuden suhteen, tuottavat malleista ihmisen käsittelyä vaativia luonnoksia, automatisoivat vain osan menetelmän vaiheista tai ovat muuten kehitystyöltään kesken tutkimuksen julkaisuajankohtana [19, 39, 44, 90, 95].

Monet modernisointimenetelmät sisältävät vaiheen käyttöliittymän rakenteen selvittämiseksi. Käyttöliittymästä tunnistetaan mahdolliset itsenäisesti päivittyvät komponentit ja näkymät, jotka muuttuvat sovelluksen suorituksen aikana [44, 48]. Esimerkiksi sovelluksen päänavigaatio- tai yläpalkkielementit ovat yhteisiä kaikille sovelluksen sivuille, ja muutokset kohdistuvat pääasiassa sivujen sisältöön. Koko sisältöosiota voidaan käsitellä yhtenä päivitettävänä komponenttina, mutta yleensä tavoitellaan tarkempaa erottelua esimerkiksi `div`-elementtien tasolla. [50, 90]

Käyttöliittymäkomponenttien tunnistamisen lisäksi menetelmissä kerätään usein tietoa sovelluksen navigoinnin rakenteesta [44, 48]. Navigointitiedon avulla voidaan selvittää, miten sovelluksen sivut ovat yhteydessä toisiinsa ja milloin sivuja näytetään. Monen sivun sovelluksissa navigointi aiheuttaa uuden sivun hakemisen palvelimelta, joten navigointirakenteen tunnistamisen yhteydessä voidaan havaita myös asiakaskomponentin suorittamat palvelinpyynnöt. [44] Linkin painallus tai lomakkeen lähetys aiheuttavat palvelinpyynnön ja siten kokonaisen sivun päivittymisen. Täten linkit ja lomakkeet vaativat erityistä huomiota modernisointiprosessissa. [19, 38, 50]

Linkkeihin ja lomakkeisiin liitetään URL-osoitteiden sijaan JavaScript-tapahtumankäsittelijät, jotka käsittelivät linkin painalluksen tai lomakkeen lähetyksen [50, 90, 95]. Tapahtumankäsittelijä tunnistaa esimerkiksi linkin painalluksen, ja suorittaa siihen liittyvän Ajax-pyynnön palvelimelle. Palvelin vastaa vain sivun muuttuneella osalla tai tarpeellisella datalla. [50, 90] Tämä mahdollistaa osittaisen sivun päivityksen suorittamisen selaimessa. Sivut kootaan selaimessa JavaScript-koodin avulla, joka päivittää DOM-puuta korvaamalla vanhat sivukohtaiset osat palvelimen palauttamilla osilla. Samalla vuorovaikutus muuttuu synkronisesta asynkroniseksi. [49] Kokonaisten sivujen päivittymisestä siirrytään näkymien ja komponenttien päivittämiseen sovelluksen tilan muuttuessa. Kokonaisten HTML-sivujen sijaan asiakkaan ja palvelimen välillä siirtyy muuttunut tila. [44]

Usein modernisoinnin yhteydessä sovelluslogiikkaa siirretään palvelimelta asiakaskomponentille [48, 89]. Tapahtumankäsittelijöihin voidaan palvelinpyyntöjen kutsujen lisäksi liittää sovelluslogiikkaa, jolloin asiakkaan on mahdollista reagoida käyttöliittymän tapahtumiin ilman yhteydenpitoa palvelimeen [39]. Esimerkiksi käyttöliittymässä näytettävän informaation suodatus on mahdollista suorittaa asiakaskomponentissa, eikä palvelinpyyntöjä tarvita. Sovelluslogiikkaa asiakaskomponenttiin siirrettäessä on kuitenkin otettava huomioon, että JavaScript-koodi on selaimessa kenen tahansa nähtävillä. Tästä johtuen sellainen sovelluslogiikka, jonka halutaan pysyvän salassa, on syytä suorittaa palvelimella. [38]

Lomakkeisiin voidaan lisätä tapahtumankäsittelijät, jotka vastaavat lomakkeen kenttien validoinnista asiakaskomponentissa. Tällöin validointi on suoritettavissa aina lomakkeen muuttuessa sen sijaan, että se suoritettaisiin palvelimella vasta lomakkeen lähetyksen jälkeen. Tästä huolimatta validointi kannattaa säilyttää myös palvelimella. [38, 95] Lähes mihin tahansa käyttöliittymän elementtiin voidaan liittää tapahtumankäsittelijä. Näin ollen käyttöliittymässä voi esiintyä tapahtumia tai asiakaskomponentti suorittaa palvelinpyyntöjä, jotka eivät suoraan liity linkkeihin, lomakkeisiin tai navigointiin. [89]

Linkkien toiminnan muuttuessa vastuu sovelluksen navigoinnista siirtyy asiakaskomponentille [48]. Navigointi perustuu sovelluksen tilan muutoksiin sivujen URL-osoitteiden sijaan. Tällöin eteen- ja taaksepäinnavigointi tai kirjanmerkit eivät enää ole suoraan käytettävissä, vaan niihin liittyvä toiminnallisuus on erikseen toteutettava kirjastojen tai rajapintojen avulla. [42, 50] Tyypillisesti sovellus avataan tietyllä URL-osoitteella, joka liittyy sovelluksen etusivuun [42]. Muiden sivujen tapauksessa yksittäinen tila on ohjelmallisesti yhdistettävä haluttuun osoitteeseen [42, 50].

Asiakaskomponentin vastuu sovelluksen toiminnasta kasvaa merkittävästi. Asiakas ei enää toimi pelkkänä pääteikkunana palvelimen valmistelemille web-sivuille. Sovellus jakautuu selkeämmin kahteen erilliseen komponenttiin, joilla on omat tehtävänsä ja jotka toimivat yhteistyössä toistensa kanssa. Tällöin sovelluksen tilanhallintaan on kiinnitettävä erityistä huomiota. [38] Perinteisissä web-sovelluksissa sovelluksen tilat ovat erillisiä

ja selkeitä. Tilat vastaavat tiettyjä palvelimen kokoamia sivuja, joilla on omat uniikit URL-osoitteensa. [42]

Yhden sivun sovelluksissa tila määräytyy dynaamisesti asynkronisten käyttöliittymän tapahtumien ja palvelinpyyntöjen seurauksena [42]. Tilan päivittämisestä vastaa JavaScript-koodi, joka suorittaa palvelinpyynnöt, käsittelee palvelimen vastaukset ja päivittää DOM-puuta. Sovelluslogiikan suorituksen aktivoivat käyttöliittymän elementteihin liitetyt tapahtumankäsittelijät. [41] Monen sivun sovelluksiin verrattuna modernisoiduissa sovelluksissa korostuvatkin käyttäjän toimien ja käyttöliittymän tapahtumien havaitseminen ja niihin reagoiminen [39]. Sovelluksen tila muuttuu tapahtumapohjaisesti käyttäjän toimien seurauksena, ja on jatkuvassa muutoksessa. [89] Tilan muutokset näkyvät ajonaikaisina muutoksina DOM-puussa. [42] Sovelluksen tilanhallinnassa on tarkoin pohdittava, miten sovelluksen kokonaistila muodostuu, miten sitä säilytetään, ja miten tila välitetään tarvittaessa asiakas- ja palvelinkomponenttien välillä. [38]

Aineiston perusteella havaitaan, että vanhentuneissa monen sivun sovelluksissa sovelluslogiikka ja sivujen kokoaminen suoritetaan pääasiassa palvelimella. Tällöin palvelin vastaa jokaiseen asiakkaan palvelinpyyntöön kokonaisella web-sivulla. Tyypillisesti modernisoinnin tavoitteena on vähentää asiakkaan ja palvelimen välistä kommunikaatiota sekä asiakkaan riippuvuutta palvelimesta. [49, 50] Usein modernisointi edellyttää JavaScript-koodin lisäämistä web-sivuihin. JavaScript suoritetaan selaimessa sovelluksen ajonaikana, ja se vastaa esimerkiksi Ajax-pyyntöjen tekemisestä. Lisäksi muutoksia voidaan tehdä suoraan palvelimen koodiin, jolloin palvelin kykenee kokonaisten sivujen sijaan palauttamaan vain sivun muuttuneen osan. [90] Toisaalta mahdollisista palvelinpuolen mallipohjista voidaan siirtyä asiakaspuolen mallipohjiin. Tällöin asiakas vastaa sivujen kokoamisesta ja pyytää palvelimelta vain muuttuneen datan esimerkiksi REST-rajapinnan kautta [38, 65]. Asiakaspuolen mallipohjia käytettäessä sovelluksen esityskerros siirtyy palvelimelta asiakkaalle. Palvelimen rooliksi muodostuu entistä selkeämmin palveluiden tarjoaminen asiakkaalle [38].

Monissa menetelmissä keskitytään kuitenkin asiakaskomponentin muokkaamiseen. Palvelimeen tehtäviin muutoksiin ei oteta kantaa, vaikka ne sovelluksen täyden modernisoinnin kannalta olisivatkin tarpeen. [39, 44, 68] Toisaalta jotkin menetelmät pyrkivät tarkoituksella välttämään suuria palvelinpuolen muutoksia, koska ne voivat osoittautua hyvin työläiksi ja aikaa vieviksi [46]. Tällöin palvelin palauttaa edelleen kokonaisia web-sivuja, vaikka yhden sivun sovelluksissa näin ei tulisi tapahtua. Ratkaisuna palvelimelle voidaan lisätä uusi yhteyskerros ja rajapinta, joka vastaa asiakkaan tekemien Ajax-pyyntöjen käsittelystä ja datan palauttamisesta oikeassa muodossa. [48] Palvelimen palauttamat kokonaiset web-sivut käsitellään asiakkaassa JavaScript-koodin avulla. Koodi lisää päivitettävälle sivulle vain muuttuneen osan sivusta. [46]

Toisinaan menetelmissä päädytään myös hybridiratkaisuihin. Sovellus voi esimerkiksi toi-

mia edelleen monen sivun sovelluksena, mutta yksittäiset sivut kykenevät päivittämään sisältöään yhden sivun sovelluksina. Tällainen lopputulos voi syntyä, mikäli automatisoitu menetelmä ei kykene muuntamaan koko sovellusta tai sovelluksen muuntaminen manuaalisesti osoittautuu liian työlääksi. [46, 49] Osa sivuista voidaan koota palvelimella, ja dynaamisempaa sisältöä vaativat sivut toteuttaa uudemmalla asiakaskomponentissa tapahtuvan renderöinnin mahdollistavalla teknologialla. Esimerkiksi React-koodia on mahdollista lisätä suoraan vanhojen sivujen sisälle. Tällöin uusia ominaisuuksia kyetään hiljalleen tuomaan sovellukseen ilman, että koko sovellusta on kerralla modernisoitava. [46] Osa tutkimuksista keskittyykin vain tiettyjen toiminnallisuuksien, kuten linkkilistojen tai lomakkeiden modernisointiin [19, 95].

Kerätyn aineiston perusteella modernisointiprosessista voidaan tunnistaa neljä merkittävää arkkitehtuurimuutosta. Sovelluslogiikkaa siirretään palvelimelta asiakkaalle, jolloin asiakaskomponentin toiminnallisuus laajenee. Kaikkien toimintojen suorittaminen ei vaadi yhteydenpitoa palvelimeen, ja asiakkaan riippuvuus palvelimesta vähenee. Tämä mahdollistaa erillisten asiakaskomponenttien toteuttamisen esimerkiksi mobiililaitteille. Asiakkaan toiminnassa korostuvat tapahtumien havaitseminen ja niihin reagoiminen. Vastuun siirtyminen asiakkaalle monimutkaistaa sovelluksen tilanhallintaa, eikä tilan säilyttäminen kokonaan palvelimella ole enää itsestäänselvää. Tyypillisesti sivujen kokoaminen siirretään palvelimelta asiakkaalle. Asiakas vastaa palvelimen palauttaman sivun osan tai datan liittämisestä oikeaan kohtaan sivua. Mallipohjia käytettäessä voidaan siirtyä palvelimen mallipohjista asiakkaan mallipohjiin. Kaikenkaikkiaan asiakaskomponentti muuttuu pääteikkunasta monimutkaiseksi toiminnaliseksi kokonaisuudeksi, joka hyödyntäää palvelimen tarjoamia palveluita.

8. YHTEENVETO

Modernisoitaessa monen sivun sovelluksia yhden sivun sovelluksiksi tavoitteena on saavuttaa työpöytäsovelluksia muistuttava käyttöliittymän interaktiivisuus ja käyttökokemus. Toisin kuin monen sivun sovelluksissa, yhden sivun sovelluksissa palvelin vastaa asiakkaan palvelinpyyntöihin vain tarvittavalla sisällöllä kokonaisen HTML-sivun sijaan. Palautettava sisältö voi muodostua esimerkiksi HTML-sivun osasta tai JSON-muotoisesta datasta. Tyypillisesti osittainen sivun päivitys toteutetaan Ajaxin avulla. Tällöin palvelinpyyntöjen suorittamisesta, palautettavan sisällön vastaanottamisesta ja sisällön lisäämisestä oikeaan kohtaan sivua vastaa selaimessa suoritettava JavaScript-koodi.

Monen sivun sovelluksissa palvelinpyyntöä edeltää yleensä käyttäjän vuorovaikutus joko linkin tai lomakkeen kanssa. Näin ollen linkkien ja lomakkeiden toiminnallisuuden muokkaaminen on merkittävässä roolissa modernisointiprosessissa. Tämän lisäksi tärkeää on tunnistaa käyttöliittymän komponentit ja näkymät, jotka ovat itsenäisesti päivitettävissä. Linkkien, lomakkeiden ja käyttöliittymäkomponenttien tunnistamiseksi modernisointimenetelmät sisältävät usein vaiheita, joissa keskitytään lähdesovelluksen navigointitiedon ja käyttöliittymän rakenteen selvittämiseen.

Monet modernisointimenetelmät ovat mallipohjaisia, ja modernisointiprosessi perustuu lähdesovelluksen pohjalta muodostettuihin malleihin ja niiden muuntamiseen. Mallien avulla modernisointi voidaan suorittaa toteutusteknologiasta riippumatta. Usein menetelmien yhteydessä esitetään algoritmi tai työkalu, joka suorittaa tarvittavat koodimuutokset, mallin muodostamisen tai koodin automaattisen generoinnin mallista. Yleensä työkalujen käyttöön liittyy kuitenkin rajoitteita tai niiden kehitystyö on artikkelin julkaisun aikaan yhä kesken.

Modernisointimenetelmien ja muun soveltuvan aineiston tarkastelun perusteella voidaan tunnistaa useita modernisointiprosessissa ilmeneviä arkkitehtuurimuutoksia. Merkittävimmät muutokset ovat asiakaskomponentissa suoritettavan sovelluslogiikan määrän kasvaminen, tapahtumapohjaisuuden korostuminen asiakkaan toiminnassa, sovelluksen tilanhallinnan monimutkaistuminen ja web-sivujen kokoamisen siirtyminen palvelimelta asiakkaalle. Aineiston haussa suurimmaksi haasteeksi nousi tuoreen tieteellisen tutkimustiedon puute. Näin ollen tulevaisuudessa kiinnostavaa voisi olla selvittää, soveltuvatko varsinkin iäkkäämmät mallipohjaiset menetelmät vanhentuneiden sovellusten muuntamisek-

si nykyaikaisiksi yhden sivun sovelluksiksi. Toisaalta voitaisiin pohtia, miten mallien muodostaminen ja käyttö on muuttunut, millaisia malleja ja suunnittelumenetelmiä on käytössä, sekä miten näitä voitaisiin hyödyntää modernisointiprosessissa.

LÄHTEET

[1] Jeremy Allaire. *Macromedia Flash MX—A next-generation rich client*. Maaliskuu 2002.

[2] *An overview of HTTP | MDN*. 4. joulukuuta 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (viitattu 09. 04. 2022).

[3] *Angular*. URL: https://angular.io/ (viitattu 09. 05. 2022).

[4] *Angular - Angular Components Overview*. 28. helmikuuta 2022. URL: https://angular.io/guide/component-overview (viitattu 09. 05. 2022).

[5] *Angular - Binding syntax*. 28. helmikuuta 2022. URL: https://angular.io/guide/binding-syntax (viitattu 09. 05. 2022).

[6] *Angular - Dependency injection in Angular*. 28. helmikuuta 2022. URL: https://angular.io/guide/dependency-injection (viitattu 09. 05. 2022).

[7] *Angular - Getting started with Angular*. 28. helmikuuta 2022. URL: https://angular.io/start (viitattu 09. 05. 2022).

[8] *Angular - Introduction to Angular concepts*. 28. helmikuuta 2022. URL: https://angular.io/guide/architecture (viitattu 09. 05. 2022).

[9] *Angular - Introduction to components and templates*. 28. helmikuuta 2022. URL: https://angular.io/guide/architecture-components (viitattu 09. 05. 2022).

[10] *Angular - Introduction to modules*. 28. helmikuuta 2022. URL: https://angular.io/guide/architecture-modules (viitattu 09. 05. 2022).

[11] *Angular - Introduction to services and dependency injection*. 28. helmikuuta 2022. URL: https://angular.io/guide/architecture-services (viitattu 09. 05. 2022).

[12] *Angular - Next steps: tools and techniques*. 28. helmikuuta 2022. URL: https://angular.io/guide/architecture-next-steps (viitattu 09. 05. 2022).

[13] *Angular - Two-way binding*. 28. helmikuuta 2022. URL: https://angular.io/guide/two-way-binding (viitattu 09. 05. 2022).

[14] *Angular - What is Angular?* 6. toukokuuta 2014. URL: https://angular.io/guide/what-is-angular (viitattu 28. 02. 2022).

[15] Tim Berners-Lee, Roy Fielding ja Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Tammikuu 2005. URL: https://www.ietf.org/rfc/rfc3986.txt (viitattu 09. 04. 2022).

[16] Alessandro Bozzon et al. ”Conceptual Modeling and Code Generation for Rich Internet Applications”. Teoksessa: *Proceedings of the 6th International Conference on Web Engineering*. ICWE ’06. Palo Alto, California, USA: Association for Computing Machinery, 2006, s. 353–360. ISBN: 1595933522.

[17] Tim Bray ja Douglas Crockford. *The JavaScript Object Notation (JSON) Data Interchange Format*. Joulukuu 2017. URL: https://www.ietf.org/rfc/rfc3986.txt (viitattu 16. 04. 2022).

[18] Jing Chen ja Bill Fisher. *Flux: An Application Architecture for React - React Blog*. 6. toukokuuta 2014. URL: https://reactjs.org/blog/2014/05/06/flux.html (viitattu 07. 05. 2022).

[19] Jason Chu ja Thomas Dean. ”Automated Migration of List Based JSP Web Pages to AJAX”. Teoksessa: *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 2008, s. 217–226.

[20] José M. Conejero et al. ”Re-engineering legacy Web applications into RIAs by aligning modernization requirements, patterns and RIA features”. *Journal of Systems and Software* 86.12 (2013), s. 2981–2994. ISSN: 0164-1212.

[21] *CSS basics | MDN*. 2. helmikuuta 2022. URL: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics (viitattu 24. 03. 2022).

[22] *ECMAScript 2023 Language Specification*. 11. huhtikuuta 2022. URL: https://tc39.es/ecma262 (viitattu 13. 04. 2022).

[23] Roy T. Fielding ja Richard N. Taylor. ”Principled Design of the Modern Web Architecture”. *ACM Trans. Internet Technol.* 2.2 (toukokuu 2002), s. 115–150. ISSN: 1533-5399.

[24] Roy Thomas Fielding. ”Architectural styles and the design of network-based software architectures”. Väitöskirja. University of California, Irvine, 2000.

[25] David Flanagan. *JavaScript: The Definitive Guide, 7th Edition*. eng. O’Reilly Media, Inc, 2020. ISBN: 9781491952016.

[26] *Flux*. URL: https://facebook.github.io/flux/ (viitattu 07. 05. 2022).

[27] Neal Ford ja Mark Richards. *Fundamentals of Software Architecture*. eng. O’Reilly Media, Inc, 2020. ISBN: 9781492043447.

[28] Piero Fraternali, Gustavo Rossi ja Fernando Sánchez-Figueroa. ”Rich Internet Applications”. *IEEE Internet Computing* 14.3 (2010), s. 9–12.

[29] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. Adaptive path. 2005.

[30] Alejandra Garrido, Gustavo Rossi ja Damiano Distante. ”Model Refactoring in Web Applications”. Teoksessa: *2007 9th IEEE International Workshop on Web Site Evolution*. 2007, s. 89–96.

[31] Dominique Guinard, Vlad Trifa ja Erik Wilde. ”A resource oriented architecture for the Web of Things”. Teoksessa: *2010 Internet of Things (IOT)*. 2010, s. 1–8.

[32] *HTML basics | MDN*. 13. maaliskuuta 2022. URL: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics (viitattu 24. 03. 2022).

[33] *HTML Standard*. 23. maaliskuuta 2022. URL: https://html.spec.whatwg.org/multipage/introduction.html (viitattu 24. 03. 2022).

[34] Mehdi Jazayeri. ”Some Trends in Web Application Development”. Teoksessa: *Future of Software Engineering (FOSE ’07)*. 2007, s. 199–213.

[35] Thilanka Kaushalya ja Indika Perera. ”Framework to Migrate AngularJS Based Legacy Web Application to React Component Architecture”. Teoksessa: *2021 Moratuwa Engineering Research Conference (MERCon)*. 2021, s. 693–698.

[36] Philip Klauzinski ja John Moore. *Mastering JavaScript Single Page Application Development*. eng. Birmingham: Packt Publishing, Limited, 2016. ISBN: 1785881647.

[37] Janne Kuuskeri ja Tommi Mikkonen. ”Partitioning Web Applications between the Server and the Client”. Teoksessa: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC ’09. Honolulu, Hawaii: Association for Computing Machinery, 2009, s. 647–652. ISBN: 9781605581668.

[38] Han Lim ja Tony Nguyen. *Migrating a Spring Web MVC application from JSP to AngularJS*. 19. elokuuta 2015. URL: https://spring.io/blog/2015/08/19/migrating-a-spring-web-mvc-application-from-jsp-to-angularjs (viitattu 06. 03. 2022).

[39] Marino Linaje, Juan Carlos Preciado ja Fernando Sanchez-Figueroa. ”Engineering Rich Internet Application User Interfaces over Legacy Web Models”. *IEEE Internet Computing* 11.6 (2007), s. 53–59.

[40] Pascal Maniraho. *The Deep Dive — Migration From Angular to React*. 10. helmikuuta 2018. URL: https://medium.com/hoopeez/the-deep-dive-migration-from-angular-to-react-ea5a807e95eb (viitattu 06. 03. 2022).

[41] Alessandro Marchetto, Paolo Tonella ja Filippo Ricca. ”Under and Over Approximation of State Models Recovered for Ajax Applications”. Teoksessa: *2010 14th European Conference on Software Maintenance and Reengineering*. 2010, s. 236–239.

[42] Ali Mesbah, Engin Bozdag ja Arie van Deursen. ”Crawling AJAX by Inferring User Interface State Changes”. Teoksessa: *2008 Eighth International Conference on Web Engineering*. 2008, s. 122–134.

[43] Ali Mesbah ja Arie van Deursen. ”An Architectural Style for Ajax”. Teoksessa: *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA’07)*. 2007, s. 9–9.

[44] Ali Mesbah ja Arie van Deursen. ”Migrating Multi-page Web Applications to Single-page AJAX Interfaces”. Teoksessa: *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*. 2007, s. 181–190.

[45] *MobX*. URL: https://mobx.js.org/ (viitattu 07. 05. 2022).

[46] Luis Neves. *How To Gradually Introduce React.js in an Old Application and migrate it to a SPA*. 14. lokakuuta 2019. URL: https://betterprogramming.pub/dont-be-afraid-of-legacy-how-to-gradually-introduce-react-js-in-a-really-old-application-79876c0dfa42 (viitattu 06. 03. 2022).

[47] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. Kesäkuu 1999. URL: https://www.ietf.org/rfc/rfc2616.txt (viitattu 09. 04. 2022).

[48] *Node.js*. URL: https://nodejs.org/en/ (viitattu 13. 02. 2022).

[49] Jaewon Oh, Woo Hyun Ahn ja Taegong Kim. ”MVC architecture driven restructuring to achieve client-side web page composition”. Teoksessa: *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. 2016, s. 45–53.

[50] Jaewon Oh et al. ”Automated Transformation of Template-Based Web Applications into Single-Page Applications”. Teoksessa: *2013 IEEE 37th Annual Computer Software and Applications Conference*. 2013, s. 292–302.

[51] L.D. Paulson. ”Building rich web applications with Ajax”. *Computer* 38.10 (2005), s. 14–17.

[52] LePage. Pete. *Storage for the web*. 11. helmikuuta 2021. URL: https://web.dev/storage-for-the-web/ (viitattu 13. 02. 2022).

[53] Arno Puder. ”A Code Migration Framework for AJAX Applications”. Teoksessa: *Distributed Applications and Interoperable Systems*. Toim. Frank Eliassen ja Alberto Montresor. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 138–151. ISBN: 978-3-540-35127-6.

[54] James Ravenscroft. *Migrating an Angular 1 Application to React*. 19. toukokuuta 2016. URL: https://medium.com/@jrwebdev/migrating-an-angular-1-application-to-react-8891ec73d462 (viitattu 06. 03. 2022).

[55] *React - A JavaScript library for building user interfaces*. URL: https://reactjs.org/ (viitattu 07. 05. 2022).

[56] *React - Add React to a Website*. URL: https://reactjs.org/docs/add-react-to-a-website.html#add-react-in-one-minute (viitattu 07. 05. 2022).

[57] *React - Components and Props*. URL: https://reactjs.org/docs/components-and-props.html (viitattu 07. 05. 2022).

[58] *React - Create a New React App*. URL: https://reactjs.org/docs/create-a-new-react-app.html (viitattu 07. 05. 2022).

[59] *React - Introducing JSX*. URL: https://reactjs.org/docs/introducing-jsx.html (viitattu 07. 05. 2022).

[60] *React - Rendering Elements*. URL: https://reactjs.org/docs/rendering-elements.html (viitattu 07. 05. 2022).

[61] *Redux - A predictable state container for JavaScript apps*. URL: https://redux.js.org/ (viitattu 07. 05. 2022).

[62] Mark Richards. *Software architecture patterns : understanding common architecture patterns and when to use them*. eng. First edition. Sebastopol, CA: O’Reilly Media, 2015. ISBN: 1-4919-7143-6.

[63] Roberto Rodríguez-Echeverría et al. ”Generating a Conceptual Representation of a Legacy Web Application”. Teoksessa: *Web Information Systems Engineering – WISE 2013*. Toim. Xuemin Lin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, s. 231–240. ISBN: 978-3-642-41154-0.

[64] Roberto Rodríguez-Echeverría et al. ”Model Driven Extraction of the Navigational Concern of Legacy Web Applications”. Teoksessa: *Current Trends in Web Engineering*. Toim. Michael Grossniklaus ja Manuel Wimmer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 56–70. ISBN: 978-3-642-35623-0.

[65] Roberto Rodríguez-Echeverría et al. ”Model-Driven Generation of a REST API from a Legacy Web Application”. Teoksessa: *Current Trends in Web Engineering*. Toim. Quan Z. Sheng ja Jesper Kjeldskov. Cham: Springer International Publishing, 2013, s. 133–147. ISBN: 978-3-319-04244-2.

[66] Roberto Rodríguez-Echeverría et al. ”Modernization of Legacy Web Applications into Rich Internet Applications”. Teoksessa: *Current Trends in Web Engineering*. Toim. Andreas Harth ja Nora Koch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 236–250. ISBN: 978-3-642-27997-3.

[67] Roberto Rodríguez-Echeverría et al. ”Re-engineering Legacy Web Applications into Rich Internet Applications”. Teoksessa: *Web Engineering*. Toim. Boualem Benatallah et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 189–203. ISBN: 978-3-642-13911-6.

[68] Gustavo Rossi et al. ”Refactoring to Rich Internet Applications. A Model-Driven Approach”. Teoksessa: *2008 Eighth International Conference on Web Engineering*. 2008, s. 1–12.

[69] Hani Samir, Eleni Stroulia ja Amr Kamel. ”Swing2Script: Migration of Java-Swing Applications to Ajax Web Applications”. Teoksessa: *14th Working Conference on Reverse Engineering (WCRE 2007)*. 2007, s. 179–188.

[70] Casimir Saternos. *Client-Server Web Apps with JavaScript and Java*. eng. First edition. Sebastopol, California: O’Reilly Media, 2014. ISBN: 1-4493-6931-6.

[71] Emmit A. Scott. *SPA design and architecture : understanding single-page web applications*. eng. 1st edition. Place of publication not identified: Manning, 2016. ISBN: 1-61729-243-5.

[72] K. Smith. ”Simplifying Ajax-style Web development”. *Computer* 39.5 (2006), s. 98–101.

[73] *Stack Overflow Developer Survey 2021*. URL: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks (viitattu 07. 05. 2022).

[74] Antero Taivalsaari ja Tommi Mikkonen. ”The Web as an Application Platform: The Saga Continues”. Teoksessa: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. 2011, s. 170–174.

[75] *The Java EE 5 Tutorial - Configuring Web Applications*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnaeo.html (viitattu 10. 05. 2022).

[76] *The Java EE 5 Tutorial - Creating Dynamic Content*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnahl.html (viitattu 10. 05. 2022).

[77] *The Java EE 5 Tutorial - Creating Static Content*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnahj.html (viitattu 10. 05. 2022).

[78] *The Java EE 5 Tutorial - JavaBeans Components*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnair.html (viitattu 10. 05. 2022).

[79] *The Java EE 5 Tutorial - JavaServer Pages Standard Tag Library*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html (viitattu 10. 05. 2022).

[80] *The Java EE 5 Tutorial - JavaServer Pages Technology*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html (viitattu 10. 05. 2022).

[81] *The Java EE 5 Tutorial - The Example JSP Pages*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnahb.html (viitattu 10. 05. 2022).

[82] *The Java EE 5 Tutorial - The Life Cycle of a JSP Page*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnahe.html (viitattu 10. 05. 2022).

[83] *The Java EE 5 Tutorial - Using Custom Tags*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnaiy.html (viitattu 10. 05. 2022).

[84] *The Java EE 5 Tutorial - Web Applications*. 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/geysj.html (viitattu 10. 05. 2022).

[85] *The Java EE 5 Tutorial - What Is a JSP Page?* 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnagy.html (viitattu 10. 05. 2022).

[86] *The Java EE 5 Tutorial - What Is a Servlet?* 2010. URL: https://docs.oracle.com/javaee/5/tutorial/doc/bnafe.html (viitattu 10. 05. 2022).

[87] Mark Thompson. *Discontinued Long Term Support for AngularJS*. 2. tammikuuta 2022. URL: https://blog.angular.io/discontinued-long-term-support-for-angularjs-cc066b82e65a (viitattu 09. 05. 2022).

[88] *TypeScript: JavaScript With Syntax For Types*. URL: https://www.typescriptlang.org/ (viitattu 09. 05. 2022).

[89] Matias Urbieta et al. ”Designing the Interface of Rich Internet Applications”. Teoksessa: *2007 Latin American Web Conference (LA-WEB 2007)*. 2007, s. 144–153.

[90] Qingling Wang et al. ”An Automatic Approach to Reengineering Common Website with AJAX”. Teoksessa: *2008 4th International Conference on Next Generation Web Services Practices*. 2008, s. 185–190. DOI: 10.1109/NWeSP.2008.34.

[91] Benoît Verhaeghe et al. ”Migrating GUI behavior: from GWT to Angular”. Teoksessa: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, s. 495–504.

[92] *What is a URL? | MDN*. 1. helmikuuta 2021. URL: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL (viitattu 09. 04. 2022).

[93] *Working with JSON | MDN*. 19. maaliskuuta 2022. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON (viitattu 16. 04. 2022).

[94] *XMLHttpRequest Standard*. URL: https://xhr.spec.whatwg.org/ (viitattu 26. 02. 2022).

[95] Ming Ying ja James Miller. ”Refactoring Traditional Forms into Ajax-enabled Forms”. Teoksessa: *2011 18th Working Conference on Reverse Engineering*. 2011, s. 367–371.