Jaakko Lindh

Reaktiivinen ohjelmointi selainpohjaisissa web-sovelluksissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotekniikka

Insinöörityö

4.5.2017

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Jaakko Lindh Reaktiivinen ohjelmointi selainpohjaisissa web-sovelluksissa 38 sivua 4.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander

Yhä suurempi osa graafisista käyttöliittymäsovelluksista on nykyään selainpohjaisia JavaScriptillä toteutettuja web-sovelluksia. Teknologian kehittyessä ja suorituskyvyn kasvaessa ne ovat monimutkaistuneet, ja nykyään esimerkiksi toimistotyökaluja, pilvipalveluna toimivaa musiikkisoitinta tai ohjelmistokehitysympäristöä voidaan suorittaa selaimessa.

Web-sovellusten monimutkaistuminen on kuitenkin aiheuttanut haasteita niiden selainpuolen ohjelmointiin. Tapahtumapohjaisen käyttöliittymän täytyy reagoida paitsi käyttäjän syötteisiin, myös palvelimelta saapuvaan dataan. Tämän kokonaisuuden hallitseminen perinteisemmillä tapahtumankäsittelymenetelmillä on osoittautunut hankalaksi.
Erääksi ratkaisuksi on ehdotettu reaktiivista ohjelmointia, joka on asynkronisiin tietovirtoihin ja niiden käsittelyyn perustuva ohjelmointiparadigma. Abstraktoimalla sovelluksen toiminnan tapahtumavirroiksi web-sovelluksia pyritään saamaan yksinkertaisemmiksi kehittää, muuttaa, debugata ja ymmärtää.

Tässä opinnäytetyössä esitellään reaktiivisen ohjelmoinnin periaatteet web-ohjelmoinnin yhteydessä sekä korkeammalla tasolla että konkreettisin esimerkein keskittyen RxJs-kirjastoon. Työssä pyritään selventämään, miksi se on lyönyt läpi juuri sillä alalla.

Käymällä läpi selain ohjelmointiympäristönä lukijalle perustellaan, miksi vanhemmat käyttöliittymäohjelmoinnin mallit on todettu siinä joissain tapauksissa vajavaisiksi. Lisäksi esitellään käytännön reaktiivista ohjelmointia siinä määrin, että ohjelmointitaitoinen lukija pystyy ottamaan sen periaatteet omissa projekteissaan käyttöön.

Avainsanat	web-ohjelmointi, tapahtumapohjainen ohjelmointi, reaktiivinen ohjelmointi

Author(s) Title Number of Pages Date	Jaakko Lindh Reactive Programming in Browser-based Web Applications 38 pages 4 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer

Today's graphical user interface programs increasingly consist of browser-based web applications written in JavaScript. As technology has evolved and browser performance has increased, these applications have been able to grow more complex and now for example office tools, a cloud based music player or an integrated development environment can be run in the browser.

This increased complexity of web applications has, however, caused challenges in their client side programming. An event driven user interface must react not only to input from the user, but often incoming data from the server as well. Managing this as a whole with traditional tools of event handling methods has proven to be difficult.

One proposed suggestion has been reactive programming, which is a programming paradigm based on asynchronous data streams and their manipulation. By abstracting the functionality of the application into streams of events the aim is to make them easier to develop, change, debug and understand.

This thesis goes through the principles of reactive programming as it relates to client side web applications, both on a higher level and with examples using the RxJs reactive programming library among others, seeking to clarify why reactive programming has broken through in this field specifically.

By describing the browser as a programming environment it is justified to the reader why older user interface programming models have been considered in it in some cases deficient. Practical reactive programming is presented to the extent that readers familiar with programming are able to begin using its principles in their own projects.

Keywords	web programming, event driven programming, reactive programming

Sisällys

Lyhenteet

Ajax	Asynchronous JavaScript and XML. Tapa tehdä selaimesta HTTP-pyyntöjä aiheuttamatta uutta sivunlatausta.
CERN	Conseil Européen pour la Recherche Nucléaire. Euroopan hiukkasfysiikan tutkimuskeskus, jossa HTTP ja HTML kehitettiin 90-luvun alussa.
CSS	Cascading Style Sheets. Kieli verkkodokumenttien ulkoasun ja tyylin määrittämiseen.
DOM	Document Object Model. Oliopohjainen rajapinta XML-dokumenttien käsittelyyn.
ECMA	European Computer Manufacturers Association. Standardointijärjestö, joka vastaa JavaScriptin spesifikaatiosta.
FRP	Functional Reactive Programming. Reaktiiviseen ohjelmointiin liittyvä ohjelmointitapa, joka keskittyy jatkuvien eikä diskreettien arvojen seuraamiseen.
HTML	Hypertext Markup Language. Kuvauskieli verkkodokumenttien rakenteen luomiseen ja linkittämiseen toisiinsa.
HTTP	Hypertext Transfer Protocol. Verkkoprotokolla, jolla asiakasohjelma kommunikoi verkkopalvelimen kanssa.
LINQ	Language Integrated Query. Microsoftin .NET-alustan datakyselyrajapinta.
MVC	Model-View-Controller. Suunnittelumalli, jota käytetään käyttöliittymäohjelmoinnissa ohjelman arkkitehtuurin jakamiseen kolmeen osaan.
REST	Representational State Transfer. Arkkitehtuuriratkaisu verkkorajapintojen suunnitteluun.
SQL	Structured Query Language. Deklaratiivinen kieli tietokantojen käsittelyyn.

URI	Uniform Resource Identifier. Merkkijono, joka määrittää yksittäisen resurssin paikan.
URL	Uniform Resource Locator. URI:n erityistapaus, jota käytetään verkossa sijaitsevan resurssin sijainnin määrittämiseen verkko-osoitteella.

1 Johdanto

Merkittävä osa tämän päivän käyttöliittymäsovelluksista on selainpohjaisia, usein palvelimen kanssa keskustelevia web-sovelluksia. Web-sovelluksilla on aiemmin tarkoitettu esimerkiksi verkkokauppaa tai keskustelupalstaa, jossa palvelin lähettää lomakepohjaisen staattisen dokumentin, jonka kautta asiakasohjelma (verkkoselain) voi lähettää palvelimelle uuden pyynnön tai navigoida toisaalle molempien aiheuttaessa uuden pyynnön ja uuden sivun latauksen sekä renderoinnin [1.] Viimeisen 10 vuoden aikana selainteknologia, päällimmäisenä niissä käytettävä ohjelmointikieli- ja ympäristö, JavaScript, on kehittynyt merkittävästi. Tästä on seurannut, että selaimessa voidaan suorittaa sovelluksia, jotka ovat perinteisesti olleet työpöytäkäyttöön paikallisesti asennettavia, kuten tekstinkäsittelysovellus tai videosoitin. Näiden selainpuolen ohjelmoinnissa haasteena on ollut asynkronisen eli aikariippumattoman tapahtumankäsittelyn ja ohjelman tilan hallinta, joka ominaisuuksien kasautuessa voi ilman varovaisuutta nopeasti muuttua hyvin vaikeaksi muuttaa, debugata, testata tai ymmärtää.

Tämän ratkaisemiseksi vuosien varrella on kehitetty useita JavaScript-ohjelmistokehyksiä, jotka abstraktoivat tilan ja tapahtumien yksityiskohtia hallitumman API:n taakse ja helpottavat ohjelmointiprosessia esimerkiksi MVC-suunnittelumallilla. Viime aikoina niissä on paradigmana saavuttanut suosiota reaktiivinen ohjelmointi [2], jonka periaatteita tämän hetken suosituimmat JavaScript-kehykset Googlen Angular 2 [3] ja Facebookin React molemmat hyödyntävät. Käsite paradigma tarkoittaa tieteen filosofiassa ajattelumallia ja lähestymistapaa ja tietojenkäsittelytieteessä sitä käytetään usein kuvaamaan ohjelmointitapaa, -kieltä tai -arkkitehtuuria [7.]

Reaktiivisen ohjelmoinnin tarkka tai yksiselitteinen määrittely on kuitenkin osoittautunut vaikeaksi ja lähteestä riippuvaiseksi [4]. Sanasta reaktiivinen tulee mieleen jokin, joka vastaa tai reagoi syötteeseen tai ärsykkeisiin – mutta toisaalta valtaosa tietokoneohjelmista täyttää tämän vaatimuksen. Käsitteitä kuten reaktiivinen ohjelmointi, funktionaalinen reaktiivinen ohjelmointi ja reaktiivinen järjestelmä käytetään usein sekaisin, eikä ole täysin selvää, ovatko kyseessä tarkat ohjelmointisäännöt vai joukko perusperiaatteita. Tietyt lähteet ovat luonteeltaan akateemisia ja toiset muistuttavat markkinointipuheita.

Tämä insinöörityö pyrkii avaamaan reaktiivisen ohjelmoinnin tarkempaa määritelmää, periaatteita ja käyttötarkoituksia erityisesti keskittyen selainpuolen web-ohjelmointiin ja siihen, miksi se on saavuttanut suosiota juuri sillä alalla sekä millaisia sille ominaisia ongelmia reaktiivinen ohjelmointi ratkaisee. Selainpuolen web-sovellusten erityishaasteiden ymmärtämistä varten on syytä tutustua myös selaimeen ohjelmointiympäristönä ja erityisesti JavaScriptin ominaisuuksiin.

2 Reaktiivinen ohjelmointi

2.1 Reaktiivisen ohjelmoinnin periaatteet

Reaktiivisen ohjelmoinnin tavoitteena on tapahtumapohjainen ohjelma, jossa muutosten vaikutus päivittyy välittömästi [5]. Muutosten päivittymisestä intuitiivinen esimerkki on taulukkolaskentaohjelma, jossa solun päivittyminen propagoituu heti kaikkiin siitä riippuviin soluihin, joista se niin ikään jatkuu kaikkiin niistä riippuviin soluihin. Esimerkiksi taskulaskin sen sijaan ei käyttöliittymänä ole kovin reaktiivinen – halutut operaatiot on suoritettava yksi kerrallaan, ja tulosten saaminen vaatii oman askeleensa.

Reaktiivisessa ohjelmoinnissa halutaan siirtyä deklaratiiviseen suuntaan imperatiivisesta tyylistä ja ajattelusta, jossa jokainen askel on jokin käsky koneelle. Deklaratiivisessa tyylissä esimerkiksi SQL-kyselyjen kaltaisesti kuvataan mitä halutaan, eikä listata yksityiskohtaisia ohjeita, miten. Korkeammalla abstraktion tasolla voidaan keskittyä enemmän kulloinkin ratkaistavaan ongelmaan eikä tietokoneen tai ohjelman yksityiskohtiin.

Reaktiivisen ohjelmoinnin pääasiallisen abstraktion voidaan katsoa olevan asynkroninen virta (data stream), joita voi olla useita ja yleensä onkin [6]. Virta on ajassa järjestetty lista tapahtumia ja ne joko voivat pitää sisällään dataa tai olla itsessään diskreettejä arvoja jostain määrätystä joukosta. Esimerkiksi tapahtumat virrassa uusista sähköpostiviesteistä voivat joko olla pelkkiä ilmoituksia (”uusi viesti saapunut”) tai ne voivat pitää sisällään sähköpostiviestin sisällön.

Virrat voivat loppua tai olla loputtomia. Avainidea on, että näitä virtoja voidaan muodostaa lähes mistä tahansa. Asemalle perjantaina saapuvat junat voivat olla tapahtumavirta, samoin hiiren kursorin koordinaatit ikkunassa. Junaesimerkki olisi

päättyvä virta, mutta kursorin koordinaatteja voidaan seurata niin kauan, kuin ikkunaa ei suljeta.

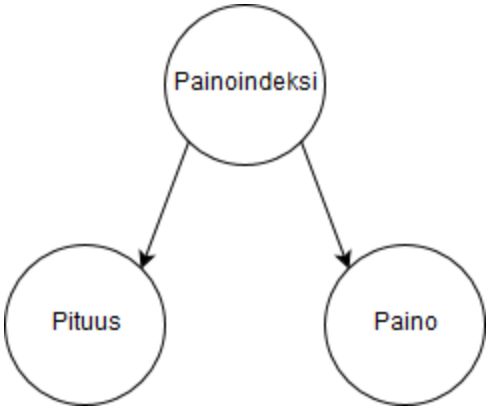
Reaktiivisen ohjelmoinnin mahdollistavat kirjastot ja teknologiat tarjoavat näiden virtojen käsittelyyn työkaluja. Reaktiivinen ohjelmointi on ohjelmointiparadigma, mutta toisin kuin esimerkiksi Java on olio-ohjelmointikieli tai Scala on sekä funktionaalinen että oliokieli, reaktiivinen ohjelmointi ei ole pääasiassa kielen ominaisuus, vaan se usein toteutetaan ohjelmakirjastolla [7.] Tällaisia kirjastoja on olemassa lähes kaikille valtavirtakielille Javasta Pythoniin.

Virtoja voidaan suodattaa, yhdistää, koota ja manipuloida monin tavoin. Kaikki virrasta riippuvat muut virrat päivittyvät uusien tapahtumien saapuessa, reaktiivisesti. Esimerkiksi edellä mainitussa junaesimerkissä jokaisen aseman voi yhdistää virraksi, jossa on koko Suomen junat. Tähän virtaan voi kohdistaa suotimen, joka jättää jäljelle vain kaksikerroksiset junat Tampereen eteläpuolisilla asemilla ja sitten luoda tästä virran, johon tulee tapahtuma joka kerta, kun 1000 matkustajaa on saapunut asemalle viimeisen 10 minuutin aikana. Virrat riippuvat niistä ylempänä hierarkiassa olevista virroista ja päivittyvät sitä mukaa kuin ylemmätkin. Juuri riippuvuudet ja riippuvuusverkot ovat toinen tapa ymmärtää reaktiivista ohjelmointia [7.]

```
pituus = 1.8
paino = 80
painoindeksi = paino / (pituus * pituus)
```

Esimerkkikoodi 1. Painoindeksin laskeminen.

Imperatiivisessa tyylissä painoindeksi voitaisiin laskea esimerkkikoodin 1 mukaisesti, jossa ohjelman tilaa muutetaan askel askeleelta.



Kuva 1. Painoindeksin laskemisen riippuvuudet

Reaktiivisessa ajattelussa ei ajatella arvojen sijoittamista muuttujiin vaan niiden riippuvuussuhteita verkossa. Esimerkiksi kuvassa 1 suunnattu graafi osoittaa painoindeksin riippuvuuden pituudesta ja painosta.

On yllättävää, minkä kaiken voikaan kuvata tapahtumavirtana. Tämä onkin reaktiivisen ajattelun perusta. Kursorista abstraktimpana versiona käyttäjän koko toiminnan käyttöliittymän kanssa voi määritellä diskreeteiksi tapahtumiksi, esimerkiksi ”tekstikentän sisältö muuttunut”, ”chat-ikkuna suljettu” ja ”videotoisto keskeytetty”. Näiden lisäksi web-palvelimet voivat lähettää käyttäjästä riippumatonta asynkronista dataa, kuten vaikkapa uudet chat-viestit tai dynaamisesti muuttuvat elokuvasuositukset. Nämä niin ikään voidaan muodostaa käsiteltävissä oleviksi tapahtumavirroiksi.

2.2 Asynkronisuus ohjelmoinnissa

Asynkronisuus tarkoittaa tässä yhteydessä jotain, joka ei tapahdu tai ole olemassa samaan aikaan, sekä operaatioita, joiden suoritusjärjestyksestä tai saapumisesta ei ole takeita eikä niiden suoritukseen voida suoraan vaikuttaa [8]. Synkronisessa suorituksessa tehtävät suoritetaan järjestyksessä ja yhden tehtävän täydellistä valmistumista odotetaan ennen seuraavan aloittamista.

Viisi pikajuoksijaa voitaisiin lähettää juoksemaan 100 metriä samalla hetkellä, ja he voisivat valmistuttuaan kukin raportoida tuloksensa, jolloin he toimisivat asynkronisesti. Jos he toimisivat synkronisesti, muiden pitäisi odottaa yhden juostessa, vaikka he voisivatkin juosta toisistaan riippumatta.

Asynkronisessa mallissa ohjelma voi tehdä samanaikaisesti jotain muuta ja siirtyä käsittelemään tehtävän tuloksia sen valmistuessa. Ohjelmakoodi on asynkronista, mikäli se aloittaa jonkin pitkän tehtävän muttei odota sen valmistumista, vaan voi aloittaa seuraavan tehtävän suorituksen välittömästi. Yleisiä esimerkkejä tällaisista pitkistä tehtävistä ovat verkkopyynnöt, levyltä lukeminen, käyttäjän syötteen odottaminen tai tarkoituksellisesti luodut viiveet. Esimerkiksi painikkeen klikkaaminen verkkosivulla on asynkroninen tapahtuma – klikkaustapahtuman ajankohdasta tai tapahtumisesta ollenkaan ei voida tietää ennalta.

Asynkronisen ohjelmoinnin haasteena synkroniseen malliin verrattuna on, että aloitettujen tehtävien valmistumisesta halutaan usein tietää. Mikäli levyltä luetaan tiedostoa, sen sisällöllä tai verkon yli tehdyn pyynnön vastauksella todennäköisesti halutaan tehdä jotain. Synkronisesti tämä olisi helppoa, sillä ohjelma ei jatku ennen tehtävän valmistumista ja voidaan olla varmoja, että siirtyessä käsittelemään tuloksia tehtävä on valmistunut. Tulosten käsittelyyn asynkronisessa mallissa on muutamia tapoja. Yleisintä on liittää asynkroniseen kutsuun takaisinkutsufunktio eli ohjeet, mitä tuloksilla tehdään, tai tapahtumapohjaisessa ohjelmoinnissa rekisteröidä kuuntelija odottamaan viestiä tehtävän valmistumisesta.

Asynkroninen ohjelmointi on siis eräs tapa saavuttaa ohjelmassa samanaikaisuutta. Samanaikaisuuden edut ovat käyttöliittymien ohjelmoinnissa erityisen tärkeitä, sillä käyttäjän kannalta ei ole ihanteellista, että koko muu ohjelma lukkiutuu sen hakiessa palvelimelta tietoa, tai että taustalla oleva laskutoimitus keskeytyy ohjelman pyytäessä käyttäjältä syötettä. Toinen tapa saavuttaa samanaikaisuutta ohjelmissa on säikeet. On huomattavaa, että nämä eivät sulje toisiaan pois – asynkronisia rakenteita on mahdollista toteuttaa säikeillä, kuten esimerkiksi C#-ohjelmointikieli tekee, mutta se ei ole välttämätöntä.

2.3 Funktionaalinen reaktiivinen ohjelmointi

Funktionaalinen reaktiivinen ohjelmointi (functional reactive programming, FRP) on usein väärinymmärretty tai liian laajasti käytetty käsite [5]. Nimityksen loivat Conal Elliott ja Paul Hudak artikkelissaan Functional Reactive Animation, joka esittelee tapahtumapohjaisen Fran-kirjaston animaatioiden luomiseen [9]. Elliottin mukaan FRP paradigmana keskittyy ajassa jatkuvasti muuttuviin arvoihin, esimerkiksi animaatiota

luodessa hahmon sijaintiin. Useat FRP:t kutsut teknologiat eivät kuitenkaan toimi näin, vaan seuraavat diskreettejä eli yksittäisiä tapahtumia, kuten käyttöliittymätapahtumia [10.]

FRP:ssä esimerkiksi lämpötilaa voitaisiin seurata jatkuvana arvona ja määrittää siitä riippuvia muita jatkuvia arvoja, kun taas virtoihin perustuvassa mallissa lämpötilan muutoksesta tulisi tapahtumavirtaan uusi tapahtuma. Digitaalisessa tietokoneessa kaikki on lopulta diskreettiä, mutta ohjelmoinnissa kyse on abstraktiosta.

Toiset lähteet taas pitävät funktionaalista reaktiivista ohjelmointia vain funktionaalisen ja reaktiivisen ohjelmoinnin leikkauksena, sillä funktionaalisen ohjelmoinnin periaatteet ovat usein siinäkin keskeisiä [7]. Funktionaalinen ohjelmointi on paradigma, jonka perusperiaatteina on, että data on muuttumatonta ja sitä käsitellään puhtailla funktioilla. Puhtaalla funktiolla ei ole niin sanottuja sivuvaikutuksia, eli se ei matemaattisen funktion tavoin vaikuta ulkomaailmaan mitenkään. Funktio, jolla on sivuvaikutus, voisi, esimerkiksi muuttaa parametriensa tai globaalin muuttujan arvoja.

Funktionaalisissa kielissä funktioita sanotaan ensimmäisen luokan (”first class”) tietotyypiksi, joka tarkoittaa, että funktiot ovat samalla viivalla muiden tietotyyppien kanssa ja funktioita voidaan niiden tavoin sijoittaa muuttujiin ja tietorakenteisiin, ne voivat palauttaa toisia funktioita sekä välittää niitä toisilleen parametreina [11]. Esimerkiksi Java-ohjelmointikielessä ei ennen sen versiota 8 näin ollut (joskin tätäkin funktiototeutusta on kritisoitu vajavaiseksi verrattuna fuktionaalisiin kieliin [35]) – funktio voi palauttaa siinä esimerkiksi merkkijonoja tai kokonaislukuja mutta ei toisia funktioita.

Funktionaalisia ohjelmia kirjoitetaan koostamalla funktioita, jotka välittävät toisilleen muuttumatonta dataa ja sen tapa hallita monimutkaisuutta näin soveltuu hyvin reaktiiviseen tyyliin. Tässä työssä käytetään nimenomaan Elliottin ”puhtaampaa” määritystä FRP:stä ja keskitytään tyyliin, jota sen mukaan pidetään pelkästään reaktiivisena ohjelmointina. Asiasta mainitaan erikseen kohdissa, joissa kyseessä on puhtaan FRP:n periaatteet.

2.4 Reaktiiviset järjestelmät

Bonér, Farley, Kuhn, ja Thompson artikkelissaan Reactive Manifesto [12] pyrkivät perustelemaan, miksi tulevaisuuden palvelinarkkitehtuurien tulee täyttää tietyt ehdot, jotta sitä voidaan kuvata reaktiiviseksi järjestelmäksi:

- Responsiivisuus: järjestelmän on vastattava nopeasti.
- Kestävyys: järjestelmä vastaa myös virhetilanteissa, ja se on suunniteltu modulaariseksi siten, että virheiden vaikutus on eristetty eri komponentteihin.
- Elastinen: järjestelmä pysyy responsiivisena erilaisen kuorman ja rasituksen alla välttäen pullonkauloja.
- Viestipohjainen (message driven): järjestelmä pohjautuu asynkroniseen viestinvälitykseen komponenttiensa välillä.

Tämä eroaa muista reaktiivisuuden määritelmistä muutamalla tavalla – se keskittyy nimenomaan palvelinarkkitehtuuriin eikä ota kantaa perusabstraktioihin kuten tapahtumavirtoihin. On kuitenkin huomionarvoista, että viestipohjainen järjestelmä muistuttaa tapahtumavirtoja monella tavalla ja että asynkronisuus on tässäkin keskeistä. Eräs nämä vaatimukset täyttävä ohjelmisto on Akka, joka on työkalupakki hajautettujen järjestelmien tekemiseen JVM (Java Virtual Machine) -alustoille ja jonka alkuperäinen tekijä on myös Reactive Manifeston kirjoittanut Jonas Bonér. Akka toteuttaa viestipohjaisen arkkitehtuurinsa actor-mallilla.

Keskityn tässä työssä tapahtumavirtoihin perustuvaan reaktiiviseen ohjelmointiin ja pidän reaktiivisen järjestelmän määritelmän sinä, miksi Reactive Manifesto sen kuvaa.

2.5 Muita reaktiiviseen ohjelmointiin liittyviä konsepteja

2.5.1 Tapahtumapohjainen ohjelmointi

Tapahtumapohjaisella ohjelmoinnilla (event-driven programming) tarkoitetaan yleensä ohjelmointitapaa, jossa ohjelman toiminta perustuu tapahtumiin reagoimiseen [31].

Erityisesti käyttöliittymäohjelmat ovat monesti tapahtumapohjaisia, jossa tapahtuma voi olla esimerkiksi ”hiiren toinen nappi alas” tai ”OK-painiketta klikattu”. Näihin tapahtumiin liitetään kuuntelijoita, jotka muuttavat ohjelman tilaa. On todettu, että tällainen koodi monimutkaistuu nopeasti niin sanotuksi spagettikoodiksi [7]. Useat tapahtumankäsittelijät muokkaavat samaa yhteistä tilaa, ja niiden koordinointi on vaikeaa. Jos ohjelmaan esimerkiksi lisää kuuntelijan, joka poistaa käyttöliittymästä painikkeen, on tarkistettava, mitkä kaikki muut kuuntelijat riippuvat kyseisestä napista. Mikäli kuuntelijoita on satoja tai tuhansia, tällaiset muutokset tulevat vaikeiksi tai mahdottomiksi tehdä. Reaktiivisessa ohjelmoinnissa nämä tapahtumat voidaan koota virroiksi ja sitä voidaan ajatella tapahtumapohjaisen ohjelmoinnin erityistapauksena, jota on esitetty ratkaisuksi juuri edellä kuvatun kaltaisiin ongelmiin.

2.5.2 Observer- ja iterator-suunnittelumallit

Vaikutusvaltaisessa kirjassa Design Patterns esitellään kaksi reaktiiviselle ohjelmoinnille keskeistä suunnittelumallia, joiden toteutus muistuttaa reaktiivista ohjelmointia ja auttaa sen ymmärtämistä. Suunnittelumalli on erityisesti olio-ohjelmoinnissa käytetty termi uudelleenkäytettävälle abstraktille tavalle toteuttaa jokin usein tarvittava ratkaisu [13.]

Observer-mallissa (to observe: seurata, valvoa) kohdeoliolla on lista seuraajistaan, jolle se ilmoittaa tilamuutoksistaan. Eräs ongelma observer-mallissa on, että seuraajat on lisättävä ja poistettava käsin, joka voi aiheuttaa muistivuotoja tai hidastaa ohjelmaa, kun osa seuraajista jää turhiksi. Reaktiivisen Reactive Extensions -kirjaston mainoslause onkin "observer-malli tehtynä oikein" ("observer pattern done right") [14.] Moni kuuntelijoihin ja tapahtumankäsittelijöihin perustuva rajapinta, esimerkiksi Javan Swing-käyttöliittymäkirjasto toteuttaa observer-mallin jollain tasolla.

Toinen tärkeä malli on iterator (to iterate: käydä läpi). Se antaa mille tahansa tietorakenteelle rajapinnan, joka palauttaa tietorakenteen seuraavan elementin järjestyksessä, tai ilmoittaa, mikäli iterator on saavuttanut lopun. Reactive Extensions yhdistää nämä mallit abstraktoimalla observer-mallin kohdeolion iteroitavaksi tapahtumalistaksi yhdistäen seuraamisen ja iteroinnin samaksi asiaksi. On huomionarvoista, että Design Patterns -kirjan kartassa toisiinsa liittyvistä malleista observeria ja iteratoria ei pidetty läheisinä.

3 Selainpohjaiset web-sovellukset

Web-ohjelmoinnissa on aina osapuolina palvelin ja asiakasohjelma, useimmiten web-selain. Palvelinpuolesta käytetään usein ilmausta backend ja selaimesta frontend. Tässä opinnäytetyössä keskitytään vain frontend-puoleen ja palvelinta pidetään mustana laatikkona, jonka kanssa selain kommunikoi HTTP-protokollalla. Esimerkiksi MVC-suunnittelumallista tai ohjelmistokehyksestä tässä yhteydessä puhuttaessa kyse on nimenomaan selain-, ei palvelinpuolesta.

3.1 HTTP

Verkkoselaimet kommunikoivat verkkopalvelinten kanssa käyttäen HTTP-protokollaa. HTTP on asiakas-palvelin-protokolla, jossa tehdään ero asiakas- ja palvelinohjelman välille – HTTP:ssä asiakas on useimmiten selain. Asiakas lähettää palvelimelle pyynnön, ja palvelin lähettää takaisin vastauksen. HTTP määrittelee palvelimen kanssa kommunikointiin metodeja, joista selaimelle tärkeimmät ovat GET ja POST.

Esimerkiksi navigoitaessa selaimella URI:hin (Uniform Resource Identifier) example.org selain tekee GET-pyynnön palvelimelle ja saa vastauksena sivun sisällön. POST-metodia taas käytetään tiedon lähettämiseen palvelimelle.

HTTP:hen liittyy läheisesti käsite REST (Representational State Transfer), joka määrittelee arkkitehtuurin HTTP:n kautta käytettävien rajapintojen suunnitteluun määrittäen, miltä URI:n resurssille (esimerkiksi verkkosivulle) tulee näyttää ja miten HTTP-metodit niitä käsittelevät [16].

Taulukko 1. Esimerkki reitistä ja HTTP-metodeista REST-arkkitehtuurissa.

	GET	**PUT**	**POST**	**DELETE**
/pelaajat/	Listaa kaikki pelaajat.	Korvaa koko pelaajakokoelma uudella.	Luo uusi pelaaja listaan.	Poista koko pelaajakokoelma.
/pelaajat/<id>	Hae yksittäinen pelaaja id:n perusteella.	Korvaa pelaaja id:llä tai luo uusi, ellei pelaajaa ole olemassa.	Ei määritelty.	Poista yksittäinen pelaaja id:n perusteella.

Esimerkiksi rajapinta, jolla käsitellään urheiluseuran pelaajia tulisi REST-arkkitehtuurin mukaan suunnitella taulukon 1 mukaan. Web-kehityksessä käytetään usein kolmansien osapuolten HTTP-rajapintoja ja on hyödyllistä, että ne on suunniteltu tietyn standardin mukaisesti.

3.2 HTML ja CSS

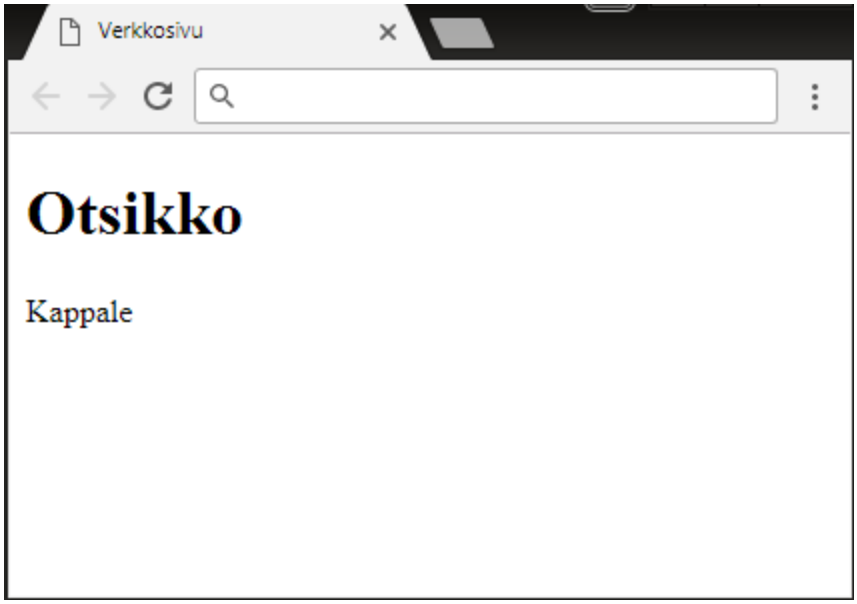
Verkkosivu on yksinkertaisimmillaan staattinen eli muuttumaton dokumentti, jollaisiksi ne alun perin oli suunniteltu. Ensimmäinen verkkosivu julkaistiin vuonna 1991 CERNissä, ja kaikki verkkosivut oikeastaan muistuttivat pitkään sitä. Niihin käytetään XML-pohjaista HTML- merkkauskieltä, jonka alkuperäinen idea oli kyetä linkittämään dokumentteja toisiinsa [15.]

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>Verkkosivu</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Otsikko</h1>
    <p id="teksti">Kappale</p>
  </body>
</html>
```

Esimerkkikoodi 2. HTML-dokumentin rakenne.

Esimerkkikoodissa 2 on validin HTML-dokumentin rakenne, josta käy ilmi, että se sisältää usein myös metadataa, tässä tapauksessa dokumentin merkistökoodauksen UTF-8. Sisennys ei ole HTML:ssä pakollista, mutta auttaa sen lukemisessa.



Kuva 2. HTML-dokumentin ulkoasu.

Kulmasulkeiden sisältämiä osioita kutsutaan elementeiksi. Esimerkiksi elementin body alkutagi on <body> ja lopputagi </body> (huomaa kauttaviiva). Elementit voivat sisältää toisia elementtejä, ja tuloksena on puumainen rakenne. HTML:ssä on useita valmiiksi määritettyjä elementtejä, jotka määräävät niille tietyn vakioulkoasun. Esimerkissä h1 tarkoittaa ensimmäisen tason otsikkoa (heading) ja p tekstikappaletta (paragraph), joka käy ilmi kuvasta 2, jossa on esimerkkikoodin 2 HTML-dokumentti renderöitynä selaimessa. Elementtien nimet sisältävät myös semanttisia vihjeitä niiden sisällöstä,

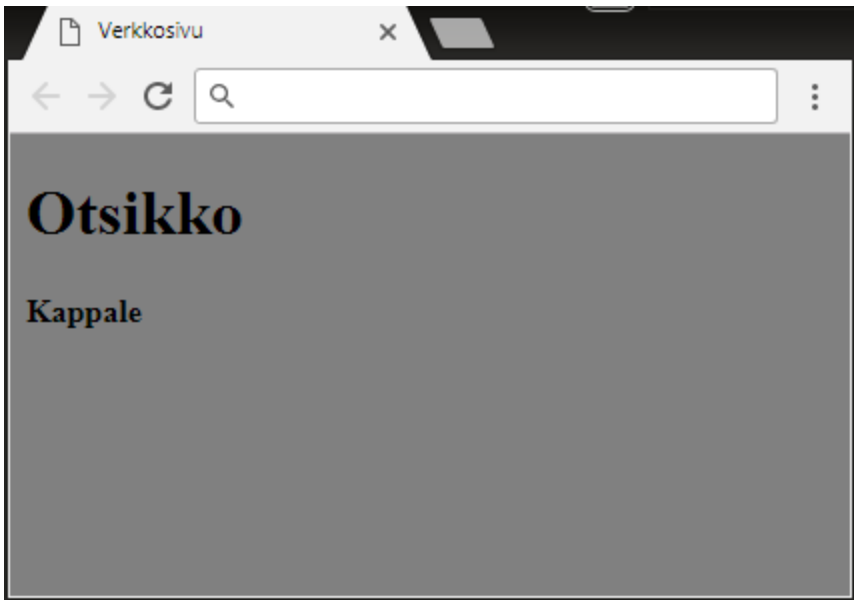
esimerkiksi p-elementin sisällön tulisi olla leipätekstiä. Muun muassa hakumoottorit ja näkövammaisten käyttämät ruudunlukuohjelmat hyödyntävät niiden tiettyä standardinmukaista käyttöä.

CSS (Cascading Style Sheets) on tapa tyylittää HTML-dokumentteja. Se koostuu valitsimista ja säännöistä, jotka muuttavat valitsimilla kohdistettujen elementtien ulkoasua. CSS pyrittiin spesifioimaan hyvin pian HTML-standardin jälkeen, jo 1996, mutta laaja selaintuki sille saatiin vasta noin vuonna 2000.

```
body {
  background-color: gray;
}
#teksti {
  font-weight: bold;
}
```

Esimerkkikoodi 3. CSS-tyylisivu.

Esimerkkikoodissa 3 käytetään elementtivalitsinta body ja id-valitsinta teksti muuttamaan sivun tausta harmaaksi sekä lihavoimaan kappale-teksti.



Kuva 3. HTML-dokumentti, johon on käytetty CSS-tyylisivua.

Kuvassa 3 näkyy CSS-tyylisivun vaikutus. Elementtivalitsin vaikuttaa kyseisen nimisiin elementteihin ja id-valitsin elementteihin, joilla on kyseinen id, huolimatta elementin nimestä.

3.3 JavaScript

Verkkosivujen mahdollisuudet ovat kehittyneet 90-luvun alun staattisista dokumenteista selaimessa suoritettaviksi täysivaltaisiksi ohjelmiksi. Edellisten esimerkkien HTML- ja CSS-koodi ei voi sisältää mitään varsinaista ohjelmalogiikkaa. Tähän tarkoitukseen on selainten tukema JavaScript-ohjelmointikieli, jolla on ohjelmarajapinta dokumenttipuuhun ja joka alun perin tehtiin Netscape-verkkoselaimeen vuonna 1995 luomaan dynaamisuutta verkkosivuihin. Sillä ei nimestään huolimatta ole tekemistä Java-ohjelmointikielen kanssa [17.]

JavaScript on dynaamisesti tyypitetty, tulkattu oliokieli, jonka syntaksi muistuttaa C:tä. Se sisältää useimmat esimerkiksi C:tä vastaavat ehtolause- ja silmukkarakenteet lähes samalla syntaksilla.

```
var muuttuja = 1;
// kommentti
for (var i = 0; i < 10; i++) {
  if (i % 2 === 0) {
    muuttuja *= 3
  }
}
```

Esimerkkikoodi 4. JavaScriptin syntaksia.

Esimerkkikoodissa 4 on käytetty aritmeettisia operaattoreita, for-silmukkaa ja if-lausetta JavaScriptillä. Siitä käy ilmi muutama JavaScriptin ominaisuus, kuten vapaaehtoinen puolipisteen käyttö lauseen jälkeen, operaattorin === käyttö yhtäläisyyden toteamiseen sekä dynaaminen tyypitys, jossa ohjelmoija ei määrittele tietotyyppejä, vaan ohjelman ajonaikainen ympäristö päättelee ne.

3.3.1 JavaScriptin oliomalli

JavaScript on oliokieli, mutta toisin kuin esimerkiksi Java tai C++, se ei ole luokka- vaan prototyyppipohjainen [18]. Luokkapohjaisessa olio-ohjelmoinnissa keskiössä ovat luokat, jotka eivät itsessään tee mitään, vaan ovat kuin sapluunoja. Niissä määritetään ominaisuudet ja toiminnallisuus oliolle, jota luokka edustaa. Luokkien pohjalta tehdään ilmentymiä, jotka ovat varsinaisia olioita. Esimerkiksi Maa-luokalla voi olla ominaisuuksina pinta-ala ja väkiluku, jonka pohjalta tehdään Suomi-olio, jolla nämä ominaisuudet ovat täytettynä. Luokat voivat periä toisia luokkia, jolloin ne sisältävät kaikki perityn luokan ominaisuudet sekä voivat määritellä niitä lisää. Luokkapohjaisessa olio-ohjelmoinnissa usein luodaan tällä tavalla luokkahierarkioita.

Prototyyppimalli ei tee eroa luokan ja olion välillä, eikä olioita tehdä luokan perusteella. Oliolla on prototyyppiolio, jolta se perii ominaisuutensa. Ne ovat luokkamallin tavoin hierarkkisia. Luokat eivät peri luokilta, vaan oliot olioilta.

Luokkapohjaisessa olio-ohjelmoinnissa olioita tehdään luokkien määrittämillä rakentajilla (constructor). Myös prototyyppimallissa voi tehdä näin, mutta esimerkiksi JavaScript-olio voi yksinkertaisimmillaan olla kokoelma avain-arvopareja.

```
var valtio = {
  laskeTiheys() {
    console.log(this.vakiluku / this.pintaAla);
  }
};
var suomi = {
  vakiluku: 5488543,
  pintaAla: 338424
};
suomi.__proto__ = valtio;
console.log(suomi.laskeTiheys());
```

Esimerkkikoodi 5. Prototyyppiperintä JavaScriptissä.

Esimerkkikoodissa 5 valtio-oliolle annetaan metodi laskeTiheys, joka viittaa this-avainsanalla sen jäseniin vakiluku ja pintaAla huolimatta siitä, ettei niitä ole valtio-oliolle määritetty. Tämä olio asetetaan suomi-olion prototyypiksi, joka on yksinkertainen

JavaScript-olio sisältäen kaksi avain-arvoparia, väkiluvun ja pinta-alan. Nyt suomi-oliolla pystyy käyttämään valtio-olion metodia ja viimeinen rivi tulostaa Suomen väestötiheyden selaimen konsoliin.

3.3.2 Funktionaalinen ohjelmointi JavaScriptillä

JavaScriptin kehittäjä Brendan Eich otti vaikutteita muun muassa funktionaalisesta Scheme-ohjelmointikielestä ja siinä on muutamia funktionaalisten kielten ominaisuuksia, kuten funktiot ensimmäisen luokan tietotyyppinä [17].

```
function teeLaskuri() {
  var x = 0;
  return function() {
    return x += 1;
  }
}
var laskuri = teeLaskuri();
laskuri(); // palauttaa 1
laskuri(); // palauttaa 2
laskuri(); // palauttaa 3
```

Esimerkkikoodi 6. JavaScriptin funktionaalisia ominaisuuksia.

Esimerkkikoodissa 6 on funktion palauttava JavaScript-funktio teeLaskuri, joka palauttaa kokonaisluvun palauttavan funktion. Esimerkistä käy ilmi JavaScriptin toinen funktionaalinen ominaisuus, sulkeumat (closure.) Se tarkoittaa, että luotu laskuri-funktio näkee ulommaisen teeLaskuri -funktion paikallisen muuttujan x arvon ja pystyy tekemään siihen muutoksia, vaikka ulommaisen funktion suoritus on jo loppunut. Tällaista näkyvyyttä kutsutaan myös leksikaaliseksi näkyvyysalueeksi (lexical scope) [11.]

3.3.3 JavaScriptin kehittyminen

JavaScriptiä käytettiin pitkään vain erittäin yksinkertaisiin tarkoituksiin, esimerkiksi kursorin automaattiseen siirtämiseen tekstikenttään hakumoottorissa eikä sillä pitkään ollut mainetta vakavasti otettavana ohjelmointikielenä [17]. Vaikka se pyrittiin

standardoimaan jo 1996 ECMAlla (European Computer Manufacturers Association) selainvalmistajat, mukaan lukien Microsoft kehittivät siitä rinnan omia versioitaan eikä voinut olettaa, että sama koodi toimisi samalla tavalla esimerkiksi Internet Explorerissa ja Mozillassa. Vasta vuonna 2005 alkoi työ standardoida selainten JavaScript-toteutukset, joka kulminoitui selainvalmistajien yhteisymmärrykseen vuonna 2008 ja ECMAScript 5-standardin kehittämiseen [19.]

Selainpuolen web-sovellusten kehittymisen on mahdollistanut JavaScript-selainsuoritusympäristöjen merkittävä nopeutuminen viimeisen 10 vuoden aikana ja juuri kielen standardointi. Kaikki selaimet eivät käytä samaa JavaScript-moottoria mutta toteuttavat pääosin saman standardin, esimerkiksi Google Chrome-selaimen JavaScript-moottorin nimi on V8, Microsoftin Internet Explorerin Chakra ja Mozillan SpiderMonkey, joka oli myös ensimmäinen JavaScript-moottori.

V8 oli ilmestyessään verrattain niin nopea, joidenkin testien mukaan jopa satoja kertoja muita nopeampi [20], että se 2009 otettiin irti selaimesta Node.js-ajonaikaiseksi ympäristöksi, jolla JavaScript-ohjelmia voidaan suorittaa palvelimilla. Node.js on yksi tämän hetken suosituimpia web-palvelinarkkitehtuureja. Chromen nopeus asetti muille selainvalmistajille paineita nopeuttaa JavaScript-moottoreitaan. Tämä kilpailu johti siihen, että yhä vaativampia ohjelmia saatettiin suorittaa selaimessa.

JavaScript on edelleen aktiivisessa kehityksessä ja ECMA jatkaa kielen kehitystä ja standardointia. Selaimiin lisätään jatkuvasti ominaisuuksia standardin seuraavista versioista (ECMAScript 6, 7 ja 8) [21.] Ohjelmoijan on kuitenkin huomattava, etteivät käyttäjät välttämättä päivitä selaimiaan niin tiheään tahtiin kuin ne saavat ominaisuuksia, vaan niiden käyttäminen edellyttää joko erilaisia yhteensopivuuskirjastoja tai uudemman JavaScript-koodin kääntämistä vanhempien selainten ymmärtämään muotoon.

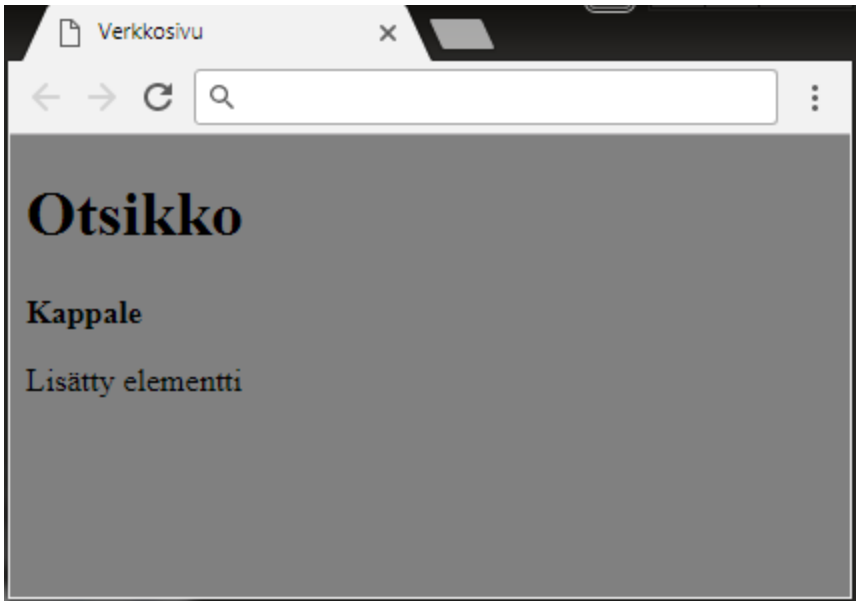
3.3.4 DOM

JavaScriptin API HTML-dokumenttiin on DOM (Document Object Model), joka on oliopohjainen tapa käsitellä XML-puuta. Selain jäsentää HTML-koodin DOM-puuksi, jota se sisäisesti käyttää. JavaScriptillä DOMia ja samalla selaimen renderöimää näkymää siitä voi käsitellä monin tavoin – poistamalla, lisäämällä tai muokkaamalla mitä hyvänsä elementtejä, liittämällä niihin kuuntelijafunktioita ja jopa muokkaamalla CSS-tyylejä [22.] DOMin ymmärtäminen on olennaista selainpuolen web-ohjelmoinnissa.

```
var div = document.createElement("div");
div.innerHTML = "Lisätty elementti";
document.body.appendChild(div);
```

Esimerkkikoodi 7. DOM:in käyttö JavaScriptillä.

Esimerkkikoodissa 7 lisätään div-elementti esimerkkikoodin 2 HTML-dokumentin tuottamaan DOM-puuhun, jossa document on sen juuri. Elementin sisältö ”Lisätty elementti” on asetettu käsin muokkaamalla sen innerHTML-kenttää.



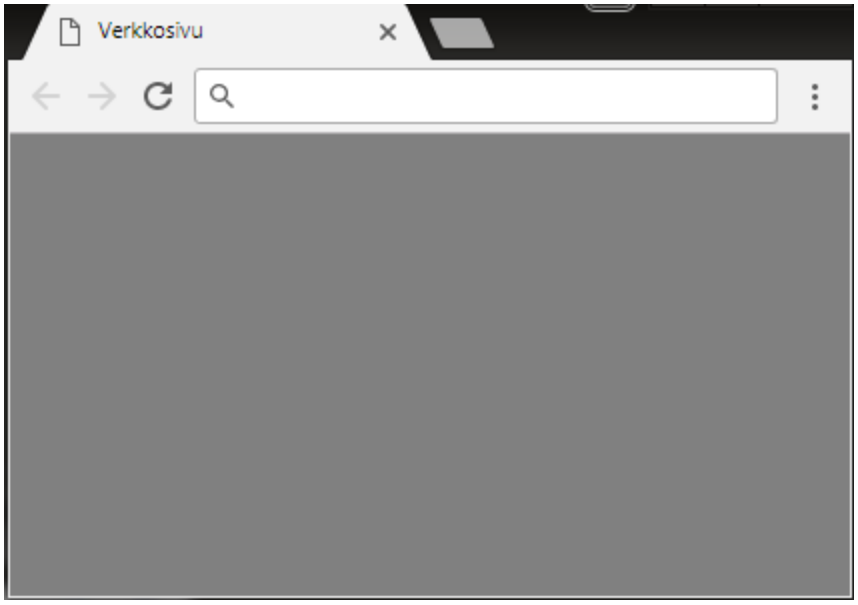
Kuva 4. Renderöity DOM-puu, johon on lisätty elementti.

Kuvassa 4 näkyy DOM-manipulaation lopputulos, jossa on lisätty div-elementti. Selaimen käyttäjälle ei ole eroa, onko elementti lisätty JavaScriptillä vai oliko se alkuperäisessä HTML-dokumentissa.

```
var body = document.body;
while (body.firstChild) {
  body.removeChild(body.firstChild);
}
```

Esimerkkikoodi 8. Elementtien poistaminen puusta DOM-operaatiolla.

Esimerkkikoodissa 8 puusta haetaan body-elementti ja sen kaikki lapsielementit poistetaan käyttäen while-silmukkaa ja DOM-funktiota removeChild. Itse body-elementti jää jäljelle.



Kuva 5. DOM-puu renderöitynä elementtien poiston jälkeen.

Kuvassa näkyy operaation lopputulos. Elementit h1, p ja lisätty div sisältöineen ovat kadonneet, koska ne kaikki olivat bodyn lapsielementtejä.

Web-sovellukset eivät välttämättä sisällä HTML:llä määritettyä rakennetta ollenkaan, vaan DOM-puu juurta lukuunottamatta luodaan JavaScriptillä, linkit lataavat sisällön dynaamisesti ja jopa selaimen osoitepalkin sisältö muutetaan antamaan illuusion, että linkkien kautta on navigoitu kuten perinteisesti. Tällaista arkkitehtuuria kutsutaan yhden sivun sovelluksiksi (single page application, SPA) [34.]

3.4 Asynkronisuus JavaScriptissä

3.4.1 Takaisinkutsut

Luvussa 2.2 määritettiin asynkronisuus ohjelmoinnissa siten, että pitkäkestoisen operaation kuten syötteen odottamisen ei ole tarpeen pysäyttää koko ohjelmaa odottamaan tuloksia, vaan ohjelman suoritus voi jatkua ja vastaus käsitellään sen

saapuessa. JavaScriptissä yleisin ratkaisu näiden tulosten käsittelyyn on takaisinkutsufunktiot (callback function) [2.]

```
function takaisinKutsu() {
  console.log("Aika täyttyi.");
}

window.setTimeout(takaisinKutsu, 5000);
console.log("Tämä tulostuu ensin.");
```

Esimerkkikoodi 9. Takaisinkutsufunktion käyttö JavaScriptissä.

Esimerkkikoodissa 9 on mahdollisimman yksinkertainen esimerkki takaisinkutsusta ja asynkronisesta suorituksesta selaimessa. JavaScriptin window.setTimeout-funktio suorittaa ensimmäisenä parametrina annetun funktion toisena parametrina millisekunneissa annetun määräajan päästä. Funktion välittäminen parametrina funktiolle on JavaScriptissä mahdollista. Esimerkin takaisinkutsu tulostaa JavaScript-konsoliin merkkijonon "Aika täyttyi". Merkkijono "Tämä tulostuu ensin" tulostetaan sen todentamiseksi, että setTimeout todella on asynkroninen funktio, eikä ohjelma jää odottamaan 5 sekunniksi.

Takaisinkutsufunktiot eivät sinänsä ole asynkronisia, vaan niitä voidaan käyttää myös synkronisesti.

```
function takaisinKutsu(x) {
  console.log(x);
}
[1, 2, 3].forEach(takaisinKutsu);
console.log("Tämä tulostuu viimeisenä. ");
```

Esimerkkikoodi 10. Takaisinkutsufunktion synkroninen käyttö.

Esimerkkikoodissa 10 JavaScriptin taulukoiden metodi forEach suorittaa takaisinkutsufunktion jokaiselle elementille järjestyksessä. Merkkijono ”Tämä tulostuu viimeisenä” tulostuu vasta, kun taulukko on käyty kokonaan läpi.

DOM-rajapinta sisältää elementtien tapahtumille, esimerkiksi hiiren klikkauksille kuuntelijoita. Näihin kuuntelijoihin on mahdollista liittää takaisinkutsuja, joka on yleinen tapa ohjelmoida web-käyttöliittymiä.

```
<body>
  <button id="painike">Nappi</button>
  <div id="laskuri">0</div>
</body>
```

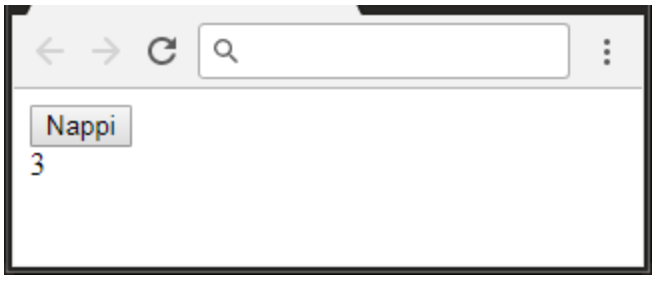
Esimerkkikoodi 11. Painike HTML-dokumentissa.

Esimerkkikoodissa 11 on button-elementti, jossa on teksti “Nappi” ja jonka id on painike, sekä div-elementti, jonka id on laskuri ja joka sisältää tekstin 0.

```
var nappi = document.getElementById("painike");
var laskuri = 0;

function takaisinKutsu() {
  laskuri += 1;
  document.getElementById("laskuri").innerHTML = laskuri;
}

nappi.addEventListener("click", takaisinKutsu);
```

Esimerkkikoodi 12. Takaisinkutsufunktio, joka liitetään button-elementtiin.

Esimerkkikoodissa 12 button-elementin click-tapahtumaan eli hiiren klikkaukseen liitetään addEventListener-funktiota käyttämällä takaisinkutsufunktio, joka lisää laskuri-muuttujan arvoon 1 ja sijoittaa sen dokumenttiin.



Kuva 6. Laskuri toiminnassa.

Kuvassa 6 nappia on painettu kolmesti, ja laskuri kasvaa joka kerralla. DOM-rajapinta sisältää useita tämänkaltaisia käyttöliittymäelementtien kuuntelijoita muun muassa kursorin siirtymiseen elementin päälle, ikkunan koon muuttumiseen tai sivun latauksen valmistumiseen. Kaikki nämä ovat luonteeltaan asynkronisia, sillä niiden tapahtumisen ajankohdasta ei tiedetä ja niitä voidaan vain kuunnella.

Yksinkertaisuudestaan huolimatta koodi sisältää takaisinkutsufunktioille ominaisen ongelman: laskuria edustava muuttuja on globaali, eli sen näkyvyys on koko ohjelman laajuinen. Globaaleja muuttujia pidetään ongelmallisena ja suuremmissa ohjelmissa niiden käyttö voi johtaa hankaluuksiin, kun usea osa koodista voi vaikuttaa samaan muuttujaan ja riippua niistä. Hyvän ohjelmointitavan mukaan data ja näkymä olisi hyvä pitää erillään ja abstraktimpaan lähestymistapaan on kehitetty MVC-kehyksiä kuten Backbone tai Angular. Takaisinkutsufunktioista huomaa, etteivät ne palauta mitään vaan tekevät jotain – tätä kutsutaan sivuvaikutukseksi. Koska ne eivät voi palauttaa mitään, tulokset on säilöttävä jonnekin.

3.4.2 Ajax

Merkittävä muutos JavaScriptiin oli Ajax (Asynchronous JavaScript and XML). Sen olennaisin osa on XMLHttpRequest-funktio, joka oli vähän käytetty tapa tehdä HTTP-pyyntöjä JavaScriptistä käsin, mutta joka oli ollut Microsoftin ja Mozillan tukema ominaisuus jo pidemmän aikaa. Sen ensimmäinen merkittävä käyttäjä oli Google GMailissa vuonna 2004 [23] ja Ajax-nimen lanseerasi Jesse James Garrett vuonna 2005 artikkelissaan Ajax: A New Approach to Web Applications. Tämä mahdollisti sen, että palvelimen kanssa kommunikointi ei vaatinut täyttä sivunlatausta, vaan esimerkiksi uudet sähköpostit saatettiin hakea dynaamisesti ja syöttää ne jo ladattuun dokumenttiin.

```
function takaisinKutsu() {
  console.log(this.responseText);
}

var xhr = new XMLHttpRequest();
xhr.addEventListener("load", takaisinKutsu);
xhr.open("GET", "http://www.example.org/");
xhr.send();
```

Esimerkkikoodi 13. XMLHttpRequest-funktion käyttö.

Esimerkkikoodissa 13 on Ajax-pyynnön käyttö yksinkertaisimmillaan. Siinä luodaan uusi XMLHttpRequest-olio, liitetään sen ”load”-tapahtumaan tuloksia käsittelemään takaisinkutsufunktio, asetetaan se antamaan HTTP GET-pyyntö haluttuun osoitteeseen ja lähetetään se. Ohjelman suoritus jatkuu ja sivun sisältö tulostetaan selaimen konsoliin HTTP-vastauksen saavuttua käyttämällä vastausolion responseText-kenttää. Mikäli takaisinkutsu unohdetaan liittää, vastaus menee hukkaan.

Eräs huomioitava seikka on, ettei XMLHttpRequestilla nimestään ole mitään erityistä tekemistä XML:n kanssa. Sillä voi pyytää mitä tahansa dokumentteja HTTP-protokollan yli, mutta yleisin tiedostonvälitysmuoto on JSON (JavaScript Object Notation), joka on syntaksiltaan kuin JavaScriptin oliot tietyin rajoittein – JSON ei muun muassa voi sisältää funktioita.

```
{
  "etunimi": "Sauli",
  "sukunimi": "Niinistö",
  "ikä": 68,
  "virat": ["puhemies", "ministeri", "presidentti"]
}
```

Esimerkkikoodi 14. JSON-tiedostomuodon syntaksia.

Esimerkkikoodissa 14 on joukko avain-arvopareja JSON-muodossa. Se voi sisältää useimpia JavaScriptin tietotyyppejä.

Yleinen asynkronisiin takaisinkutsuihin liittyvä ongelma on usean takaisinkutsun liittäminen toisiinsa.

```
document.getElementById("painike").click(function() {
          $.get("/posts", (function(tulokset) {
            tulokset.forEach(function(tulos) {
                    console.log(tulos);
            })
          }));
});
```

Esimerkkikoodi 15. Sisäkkäisiä takaisinkutsuja.

Esimerkkikoodissa 15 on kolme sisäkkäistä takaisinkutsua: kuuntelija napinpainallukselle, Ajax-kutsu ja forEach tulosten käsittelylle. Esimerkki on hankala lukea eikä siitä ole välittömästi selvää, mitä koodi tekee. Yksinkertaisuuden vuoksi Ajax-kutsussa käytetään suositun JQuery-kirjaston $.get-funktiota esimerkkikoodin 13 pidemmän version sijasta. Sen lisäksi takaisinkutsussa on siirrytty käyttämään JavaScriptin funktioliteraaleja, jossa funktion runko kirjoitetaan suoraan toisen funktion argumentteihin, ilman sen nimeämistä sijoittamalla se muuttujaan.

Erään useita JavaScript-ohjelmia analysoineen tutkimuksen mukaan suurin osa takaisinkutsuista on tähän tapaan sekä sisäkkäisiä että funktioliteraaleja käyttäviä [24]. Tällaisen JavaScript-koodin välttämisestä on kirjoitettu paljon ja useista sisäkkäisistä takaisinkutsuista käytetään nimitystä ”callback hell”. Reaktiivinen ohjelmointi on yksi esitetty vaihtoehto tämän monimutkaisuuden hallintaan.

3.4.3 Lupaukset (promise)

ES6-standardissa JavaScriptiin lisättiin takaisinkutsuille vaihtoehtoinen ja monesti parempi tapa hallita asynkronisuutta. Lupaus on arvo, joka edustaa jotain tuntemattomassa ajankohdassa suoritettavan operaation tulosta, eli se itsessään abstraktoi kyseisen operaation sekä pitää sisällään sen tuloksen. Tästä seuraa niiden tärkeä ominaisuus, että tulos tulee otettua talteen ilman, että mihinkään on liitetty erikseen kuuntelijaa – toisin kuin takaisinkutsumallissa, jossa esimerkiksi XMLHttpRequestin arvo on hukattu, mikäli se on lähetetty unohtaen liittää tulosten käsittelyyn takaisinkutsu [25.]

```
var lupaus = new Promise(function(resolve, reject) {
   setTimeout(function() { resolve("Aika täyttyi."); }, 5000);
});
lupaus.then(console.log);
```

Esimerkkikoodi 16. Lupauksen käyttö JavaScriptissä.

Esimerkkikoodissa 16 käytetään asynkronista funktiota setTimeout palauttamaan merkkijono 5 sekunnin jälkeen. Uusi lupaus tehdään antamalla Promise-olion rakentajalle funktio, jolla on kaksi käsittelijää, joiden nimet yleisesti ovat resolve

(ratkaise) ja reject (hylkää). Asynkroninen operaatio suoritetaan tämän olion sisällä ja resolve-käsittelijää kutsutaan palauttamaan operaation arvo sen valmistuttua, tässä tapauksessa merkkijono ”Aika täyttyi” 5 sekunnin kuluttua. Reject-käsittelijää ei tässä esimerkissä käytetä.

Lupauksen then-metodia käytetään sen arvon eli sen sisältämän operaation tuloksen käsittelyyn. Tässä tapauksessa sille annetaan yksiparametrinen console.log-funktio, jolla tulos printataan selaimen konsoliin. Metodin voi liittää lupaukseen välittömästi, jolloin sitä kutsutaan lupauksen valmistuttua, tai sitä voidaan kutsua vasta tässä tapauksessa yli 5 sekunnin kuluttua lupauksen luomisesta, jolloin se niin ikään tulostaa lupauksen arvon – lupaus pitää sisällään operaation tuloksen.

Lupaukset voivat olla kolmessa tilassa, joita ovat pending (odottaa operaation tulosta), fulfilled (täytetty, kun ensimmäistä käsittelijää on kutsuttu), tai rejected (hylätty, kun toista käsittelijää on kutsuttu.) Reject-käsittelijää käytettäisiin ilmaisemaan operaation epäonnistuminen, esimerkiksi sijoittamalla lupauksen arvoksi jokin virhekoodi. Esimerkkikoodin 16 lupaus on tilassa pending ennen 5 sekunnin kulumista eikä se sisällä vielä arvoa. 5 sekunnin jälkeen sen tila on fulfilled, ja koska sillä ei ole reject-käsittelijää, se ei voi olla rejected-tilassa.

Lupauksen pääasiallinen ero takaisinkutsuihin nähden on siis se, miten se käsittelee asynkronisen operaation tuloksen. Koska takaisinkutsufunktio ei voi palauttaa mitään, sen on säilyttävä tulos johonkin. Lupaus sen sijaan pitää tuloksen sisällään ja sen voi ajatella edustavan kyseistä arvoa, esimerkkikoodissa merkkijonoa. Lupaukset ovat esimerkki uudesta ominaisuudesta, joille ei ole täydellistä selaintukea ja joka on otettava huomioon – Firefox ja Chrome ovat tukeneet niitä noin vuodesta 2014 lähtien.

3.5 JavaScriptiksi käännettävät kielet

Selaimen JavaScript-ympäristöjen nopeus ja tehokkuus on aiheuttanut sen, ettei muiden ohjelmointikielten suosijoiden ole järkevää pyrkiä lisäämään lisää kieliä selaimeen, vaan tehdä JavaScriptia tuottavia kääntäjiä joko olemassa oleviin kieliin tai kokonaan uusiin. Tällaisia kääntäjiä on lukuisia, muutamana esimerkkinä CoffeeScript, TypeScript, Scala.js, ClojureScript ja Elm [26.] Useat frontend-kehitysympäristöt ”kääntävät” jopa JavaScriptin JavaScriptiksi. Tämän etu on, että ohjelmoija voi käyttää uudempien

JavaScript-versioiden ominaisuuksia kuin mitä selaimet tukevat, mutta myös koodin optimointi sekä rakenteellisesti kuten esimerkiksi C-kääntäjä tekee, että myös pienemmiksi latauksiksi poistamalla esimerkiksi kommentit [27.] Tätä prosessia kutsutaan myös minifioinniksi (minification.)

3.6 WebSocketit

WebSocket on vuonna 2011 standardoitu teknologia, joka mahdollistaa kaksisuuntaisen (full duplex) kommunikaation palvelimen ja selaimen välillä yhden TCP-yhteyden yli. Se ero HTTP:hen on, että HTTP on tilaton ja asiakkaan pyynnön ja palvelimen vastauksen jälkeen kaikki seuraava kommunikaatio hoidetaan uusilla HTTP-pyynnöillä [28.] Tämä soveltuu huonosti tiettyihin reaaliaikaisiin sovelluksiin, esimerkiksi chat-huoneeseen, jossa keskitetty palvelin lähettää jokaiselle huoneessa olijalle uudet viestit heti niiden saapuessa. HTTP-ratkaisulla asiakkaiden pitäisi lähettää palvelimelle jonkinlainen ajastettu kysely uusista viesteistä reaaliaikaisuuden tuntuman saavuttamiseksi melko usein.

Mitä jos käyttäjiä on tuhansia tai satojatuhansia? Ongelmaksi muodostuu, että suuri osa liikenteestä on HTTP:n metadataa ja jokaiseen yhteyteen pitää tehdä uusi TCP-kädenpuristus. WebSocket ratkaisee ongelman palvelimen ylläpitämällä jokaiseen asiakkaaseen yhteyttä, voiden itse lähettää niille dataa ilman erillistä pyyntöä vähäisellä overheadilla. Tuloksena olevaa asynkronista datavirtaa voi hyödyntää reaktiivisessa ohjelmoinnissa. WebSocketeilla on nykyään laaja selain- ja palvelintuki.

3.7 Reaktiivisen ohjelmoinnin mahdollistavia selainkirjastoja ja –teknologioita

3.7.1 Reactive Extensions

Reactive Extensions (Rx) on alunperin Microsoftin vuonna 2009 julkaisema kirjasto .NET-alustan kieliin (C#, VB.NET.) Sen voi ajatella yhdistävän observer- ja iterator-suunnittelumallit observable-datavirtojen toteutukseen. Rx:stä on sittemmin tehty versiot usealle ohjelmointikielelle (esimerkiksi Java ja C++) ja tässä työssä käytetään esimerkkeihin sen JavaScript-versiota RxJs. Reactive Extensionsia käytetään teollisuudessa laajalti, muutamia sen käyttäjiä ovat Netflix, Microsoft, GitHub ja Trello.

Seuraavassa luvussa käydään RxJs:ää läpi tarkemmin reaktiivisen ohjelmoinnin esimerkkien yhteydessä.

3.7.2 React

React on Facebookin vuonna 2013 julkaisema JavaScript-kirjasto käyttöliittymien tekemiseen. Se on erittäin suosittu, ja sitä Facebookin lisäksi tällä hetkellä käyttää muun muassa Pinterest, Twitter, Atlassian ja Airbnb. Reactin filosofia on, että käyttöliittymiä koostetaan komponenteista ja niiden puumaisista hierarkioista, joista mahdollisimman moni pyritään tekemään tilattomaksi, eli ne ovat kuin puhtaita funktioita jotka samoilla argumenteilla tuottavat aina saman tuloksen. Komponentit voivat olla uudelleenkäytettäviä ja ne muistuttavat MVC-kehysten templateja, mutta voivat myös sisältää tilaa ja logiikkaa. React-komponenttien kirjoitukseen käytetään erityistä XML:ää muistuttavaa JSX-syntaksia [29.]

React-komponenttien suunnitteluperiaate on yksisuuntainen tietovuo, joka tarkoittaa, etteivät komponentit voi muokata parametrejaan ja esimerkiksi Angular 1:n tai MVC:n tapaan vaikuttaa malliin (kaksisuuntainen tietovuo), vaan ne kommunikoivat emokomponenteilleen takaisinkutsufunktioilla. Muutoksien tapahtuessa koko komponenttipuu renderoidaan tarpeellisilta osiltaan uusiksi. React ei muokkaa DOM-puuta suoraan vaan sisältää virtuaalisen DOMin, jota diff (difference) -algoritmilla verrataan jokaisella muutoksella edelliseen, ja varsinaiseen selaimen DOMiin kohdistetaan vain minimaaliset muutokset, joita voi olla vain yksi merkki hakukentässä. Reactista on olemassa lisäksi React Native-versio, jolla Android- ja iOS-sovellusten käyttöliittymiä voidaan tehdä JavaScriptillä Reactin periaattein.

React ei nimestään huolimatta ollenkaan sido ohjelmoijaa minkäänlaiseen reaktiiviseen tyyliin tai ohjelmointitapaan, eikä oikeastaan tarjoa primitiivejä, kuten esimerkiksi RxJs. Se antaa vain kirjaston toteuttamaan osan käyttöliittymää, jota MVC-mallissa vastaisi näkymä. React kuitenkin tarjoaa erinomaisen näkymätoteutuksen, jonka voi yhdistää reaktiivisen ohjelmoinnin periaatteisiin. Esimerkiksi komponenttien takaisinkutsufunktiot voivat vastata tietovirtamallin tapahtumia.

Reactia käytetään usein jonkin muun kirjaston tai arkkitehtuurillisen ratkaisun kanssa, joka ohjaa ohjelmoijaa hyväksi todettuihin tapoihin. Yksi tällainen arkkitehtuuri on Redux [30], joka selkeästi erottaa käyttöliittymässä tilan tietovarastoksi (MVC:ssä malli) ja

rajoittaa sen käsittelyn hyvin määritettyihin tapahtumiin jotka tulevat em. takaisinkutsufunktioista, itse komponenttien ollessa tilattomia. Näin koko käyttäjän interaktio käyttöliittymän kanssa voidaan kuvata listalla tai virralla tapahtumista. Tämä helpottaa debuggaamista huomattavasti, kun tapahtumat käyttöliittymän mihin tahansa tilaan saattamiseen voidaan listata ja niitä voidaan esimerkiksi "kelata takaperin".

3.7.3 Elm

Elm on funktionaalinen täsmäkieli (domain specific language) frontendin web-ohjelmointiin, eli se on kielenä tehty yksinomaan tätä tarkoitusta varten. Elmin kehitti Evan Czaplicki opinnäytetyössään Elm: Concurrent FRP for Functional GUIs [31.] Se on JavaScriptiksi käännettävä kieli ja sen syntaksi muistuttaa suurilta osin puhtaasti funktionaalista, vahvasti tyypitettyä Haskell-kieltä, jolla sen kääntäjä on kirjoitettu. Koko frontend-sovellus kirjoitetaan Elmillä ja käännetään JavaScript, HTML- ja CSS-tiedostoiksi.

Elm kehitettiin ottamaan kantaa tiettyihin senhetkisen reaktiivisen käyttöliittymäohjelmoinnin ongelmiin, kuten viiveisiin, kun käyttöliittymä ei välittömästi pysty vastaamaan, sekä tarpeettomaan uudelleenlaskentaan, jossa koko riippuvuusverkkoon kohdistuu toisinaan tarpeettomia muutoksia. Jälkimmäisen ongelman Elm ratkaisee memoisaatiolla, joka funktionaalisessa ohjelmoinnissa tarkoittaa, että puhdas funktio voi muistaa aiemmilla samoilla parametreilla suorittamansa rungon, sillä sen tulos on aina sama. On huomattavaa, että käsite FRP tässä tapauksessa ei vastaa Elliottin määritelmää puhtaasta FRP:stä.

Elmin tärkein ominaisuus on sen arkkitehtuuri, jolla se pakottaa ohjelmoijan kirjoittamaan sovelluksen. Elm-arkkitehtuurissa on

- malli, joka on koko sovelluksen kaikki tila.
- päivitysfunktio, jolla sovelluksen tilasta tuotetaan sen seuraava tila.
- näkymä, jolla ohjelman tilasta tuotetaan HTML:ää.

Näkymä tuottaa signaaleja, jotka päivitysfunktio muuttaa tilamuutoksiksi. Arkkitehtuuri muistuttaa vahvasti edellä mainittua Reduxia, johon otettiin vaikutteita Elmistä.

Seuraavaksi esitellään luvun 2 kuvan 1 painoindeksilaskuri toteutettuna Elm-arkkitehtuurilla.

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import String
```

Esimerkkikoodi 17. Elmin standardikirjaston funktioita tuodaan käytettäväksi.

Esimerkkikoodissa 17 Elmin standardikirjastosta tuodaan rajapintoja DOM-toimintoihin. Se täsmäkielenä sisältää kaiken tarpeellisen frontend-sovellusten ohjelmointiin.

```
type alias Model = { pituus : Int, paino : Int }
model : Model
model = { pituus = 0, paino = 0 }
```

Esimerkkikoodi 18. Mallin määrittely.

Esimerkkikoodissa 18 malliksi määritetään kaksi kokonaislukua (pituus ja paino) sisältävä tietorakenne ja tämän tyypin nimeksi (type alias) annetaan Model. Nämä kaksi kokonaislukua ovat koko ohjelman kaikki tila. Tämän jälkeen luodaan uusi Model-tietorakenne alustaen se arvoilla 0. Kaksoispisteellä ennen alustusta kerrotaan seuraavan sijoituksen tyyppi. Tätä kutsutaan tyyppiannotaatioksi, joka ei ole pakollinen mutta toimii vihjeenä kääntäjälle ja auttaa dokumentoimaan koodia.

```
type Msg
  = MuutaPituutta String
  | MuutaPainoa String
```

Esimerkkikoodi 19. Viestin määrittely.

Esimerkkikoodissa 19 määritetään Elm-arkkitehtuurin mukainen viesti, joita näkymä tuottaa. Syntaksi tarkoittaa, että tyyppi Msg voi olla joko MuutaPituutta tai MuutaPainoa. Molemmat pitävät sisällään uutta arvoa edustavan merkkijonon.

```
update : Msg -> Model -> Model
update msg model =
```

```
  case msg of
    MuutaPituutta uusiPituus ->
      { model | pituus = Result.withDefault 0 (String.toInt uusiPituus)
}
    MuutaPainoa uusiPaino ->
      { model | paino = Result.withDefault 0 (String.toInt uusiPaino) }
```

Esimerkkikoodi 20. Päivitysfunktion määrittely.

Esimerkkikoodissa 20 tyyppiannotaatio Msg -> Model -> Model tarkoittaa, että päivitysfunktio update ottaa vastaan viestin, vanhan mallin ja palauttaa uuden mallin. Viesti voi olla MuutaPituutta tai MuutaPainoa. Päivitysfunktio käyttää kielen hahmonsovitusta (pattern matching), joka yksikäsitteisesti tekee viestin kummallekin mahdolliselle arvolle oikean mallipäivityksen. Hahmonsovituksen voi ymmärtää tehokkaampana switch-lauseena, joka varmistaa, että jokainen mahdollinen tapaus tulee käsiteltyä. Jos esimerkiksi MuutaPainoa-tapauksen poistaisi, ohjelma ei menisi kääntäjästä läpi.

Tapausten käsittelijässä syntaksi { model | ... } tarkoittaa, että vastaanotetusta tietorakenteesta model palautetaan uusi versio, jossa putkimerkin jälkeiset kentät ovat muutettu halutunlaiseksi ja kummassakin tapauksessa mallista muutetaan vain asianmukainen kenttä eli joko pituus tai paino. Arvoille tehdään tietotyypinmuutos, koska HTML-elementteihin voi sijoittaa Elmissä vain merkkijonoja. Result.withDefault 0 tarkoittaa, että mikäli tyypinmuutos ei onnistu, vakioarvoksi asetetaan 0.

```
laskeBMI : Int -> Int -> Float
laskeBMI pituus paino =
  toFloat paino / ((toFloat pituus / 100) ^ 2)
```

Esimerkkikoodi 21. Funktio painoindeksin laskemiseen.

Esimerkkikoodissa 21 on apufunktio painoindeksin laskemiseen. Tyyppiannotaatio Int -> Int -> Float tarkoittaa, että funktio ottaa vastaan kaksi kokonaislukua ja palauttaa liukuluvun. Tyypinmuunnokset toFloat-funktiolla ovat pakollisia, sillä Elm ei vahvasti tyypitettynä kielenä tee automaattista muunnosta kokonais- ja liukulukujen välillä.

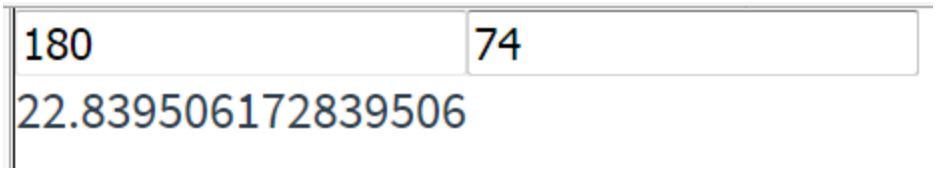
```
view : Model -> Html Msg
```

```
view model =
div []
[ input [ placeholder "Pituus", onInput MuutaPituutta ] []
, input [ placeholder "Paino", onInput MuutaPainoa ] []
, div [] [ text (toString (laskeBMI model.pituus model.paino)) ]
]
main = beginnerProgram { model = model, view = view, update = update }
```

Esimerkkikoodi 22. Elm-arkkitehtuurin näkymäfunktio.

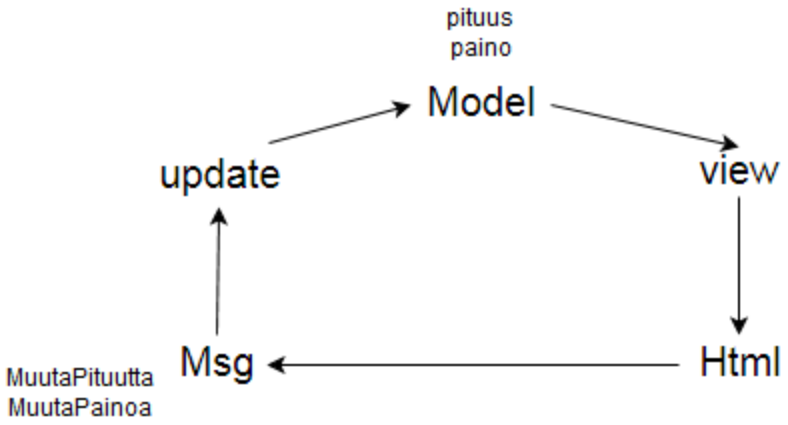
Esimerkkikoodissa 22 on näkymäfunktio, joka tyyppiannotaationsa perusteella ottaa vastaan mallin ja palauttaa sen HTML-vastineen. Nuolen oikealla puolella oleva tyyppi Html Msg tarkoittaa, että tyyppi on Html, joka tuottaa Msg-tyyppejä, eli tässä tapauksessa määritettyjä MuutaPituutta- ja MuutaPainoa-viestejä. HTML-puun rakenne luodaan käyttämällä Elmin HTML-funktioita, joissa elementtityyppi kuten div on funktion nimi, sen ensimmäinen argumentti on lista HTML-attribuuteista kuten div tai placeholder ja toinen argumentti lista sen lapsielementeistä.

Sivun input-kenttiin liitetään onInput-funktiot, jotka lähettävät niille kuuluvat viestit aina kentän arvon muuttuessa. Viestit muuttavat mallia, ja mallin muutokset heijastuvat reaktiivisesti malliin, josta se päivittyy tuloskentän sisältävään div-elementtiin. Lopuksi malli, näkymä ja päivitysfunktio kootaan yhteen ohjelmaksi beginnerProgram-funktiolla.



Kuva 7. Painoindeksiohjelma toiminnassa.

Koko ohjelma on tässä ja Elm-kääntäjä tuottaa siitä kaiken tarvittavan HTML:n ja JavaScriptin. Sen lopputulos näkyy kuvassa 7. Kaikki Elm-ohjelmat toteuttavat saman yksinkertaisen, mutta erittäin käyttökelpoisen arkkitehtuurin.



Kuva 8. Painoindeksisovelluksen toiminta.

Kuvassa 8 havainnollistetaan Elm-arkkitehtuurin toiminta painoindeksilaskurin tapauksessa. Html tuottaa Msg-tyyppejä, jotka voivat olla MuutaPituutta tai MuutaPainoa. Päivitysfunktio tekee malliin määritellyt muutokset, joka heijastuu näkymään automaattisesti. Näkymän Html tuottaa jälleen uusia Msg-tyyppejä.

4 Esimerkkejä virtojen käsittelystä RxJs:llä

Tässä luvussa esitellään reaktiivisen ohjelmoinnin primitiivejä eli perusosasia, joilla reaktiiviseen tyyliin ohjelmoidaan. Esimerkeissä käytetään Reactive Extensionsin JavaScript-versiota RxJs:n versiota 5, mutta ne ovat melko samanlaisia eri kirjastoissa, esimerkiksi Flapjaxissa ja Bacon.js:ssä jotka niin ikään ovat reaktiivisia JavaScript-kirjastoja. Monet primitiiveistä muistuttavat muistakin yhteyksistä, esimerkiksi C#:n LINQ:sta tuttuja listaoperaatioita, mikä ei ole sattumaa – tapahtumavirtoja ja kokoelmia voidaan käsittää samana asiana.

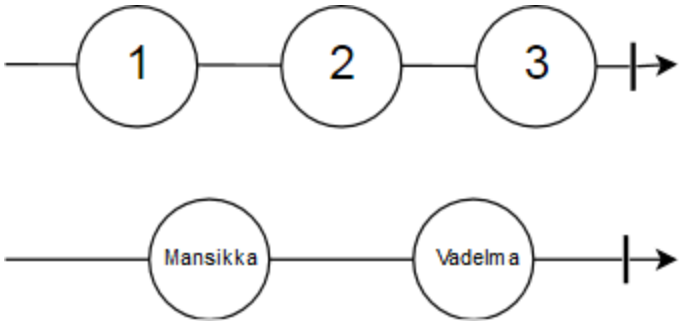
4.1 Virtojen luominen

Rx:n tärkein abstraktio on Observable, joka edustaa tapahtuma- tai tietovirtaa. Ne ovat muokkaamattomia, eikä niiden elementtejä voi käsin lisätä tai poistaa. Observablen loppuun saapuu lisää elementtejä sen lähteestä. Niitä voidaan luoda lähes mistä tahansa – DOMin tapahtumakuuntelijoista, Ajax-kutsuista tai tavallisista JavaScriptin staattisista listoista, jopa muuttujista.

```
var luvut = Rx.Observable.of(1, 2, 3);
var marjat = Rx.Observable.from(["Mansikka", "Vadelma"]);
```

Esimerkkikoodi 23. Uuden Observablen luominen.

Esimerkkikoodissa 23 luodaan kaksi Observablea kokonaisluvuista ja merkkijonotaulukosta. ReactiveX käyttää dokumentaatiossaan virtojen visualisointiin niin sanottuja marble-diagrammeja, joiden nimi tulee niissä marmorikuulia muistuttavista elementeistä virrassa.



Kuva 9. Marble-diagrammi.

Kuvassa 9 on esimerkkikoodin 23 Observablet marble-diagrammeina. Aika juoksee vasemmalta oikealle nuolen suuntaan, ympyrät ovat elementtejä ja pystyviiva merkitsee virran loppua. Molemmat ovat päättyviä virtoja. Päättymättömässä virrassa pystyviivaa ei merkittäisi.

```
var painike = document.getElementById("painike");
var klikkaukset = Rx.Observable.fromEvent(painike, "click");
```

Esimerkkikoodi 24. Observablen luonti DOM-tapahtumasta.

Esimerkkikoodissa 24 luodaan Observable klikkaustapahtumista DOM-elementille. Tähän virtaan saapuu jokaisesta hiiren klikkauksesta click-tapahtuma, joka sisältää metadataa elementistä ja klikkauksesta, esimerkiksi elementin nimen, koordinaatit ja millä hiiren painikkeella elementtiä klikattiin.

Kuva 10. Klikkaustapahtumien virta.

Kuvasta 10 käy ilmi, että kyseessä on päättymätön virta. Klikkausten kuuntelua ei käsin lopeteta eikä virtaa suljeta – se on olemassa, kunnes se tuhotaan esimerkiksi sulkemalla selainikkuna. Visualisaatiot staattisista kokoelmista ja oikeasta asynkronisuuden lähteestä kuten klikkauksista näyttävät samalta. Niitä myös voidaan käsitellä täysin samoin tavoin, ja eräs Reactive Extensionsin perusperiaate on, että virrat ja kokoelmat ovat sama asia.

```
Rx.Observable.fromPromise(fetch("http://example.org/"))
```

Esimerkkikoodi 25. Observablen tekeminen lupauksesta.

Myös luvussa 3.4.3 kuvatusta lupauksesta voi tehdä Observablen. Esimerkkikoodin 25 Observable käyttää fetch-rajapintaa Ajax-pyynnön tekemiseen ja Observable emittoi lupauksen tuloksen tai virheviestin sen valmistuessa. Emittoiminen tarkoittaa samaa asiaa kuin uuden elementin virtaan saapuminen.

4.2 Virtojen kuuntelu

Luoduilla virroilla halutaan usein tehdä jotain. Rx:ssä Observableja kuunnellaan antamalla niiden subscribe-metodille takaisinkutsufunktio. Tätä prosessia kutsutaan Observablen tilaamiseksi (subscribe).

```
luvut.subscribe(console.log);
marjat.subscribe(console.log);
klikkaukset.subscribe(console.log);
```

Esimerkkikoodi 26. Observablen tilaaminen.

Esimerkkikoodissa 26 esimerkkikoodin 23 ja 24 Observableihin liitetään tilaus. Annettu takaisinkutsu suoritetaan Observablen emittoidessa jotain. Tässä käytetään yksiargumenttista selaimen console.log-funktiota, joka kahdelle ensimmäiselle Observablelle tulostaa kunkin elementin arvon ja viimeiselle tulostaa klikkauksen metadataolion jokaisella klikkauksella.

Observablet voivat emittoida kolmenlaisia arvoja. Elementtiensä arvon lisäksi ne voivat emittoida signaalin virhetilanteelle ja virran loppumiselle.

```
marjat.subscribe(console.log, x => console.log("Virhe"), y => console.log("Loppu"));
```

Esimerkkikoodi 27. Observablen kaikkien arvotyyppien tilaaminen.

Käsittelijät näille annetaan subscribe-metodille esimerkkikoodin 27 mukaisessa järjestyksessä. Esimerkkikoodin 23 Observablella on loppu, joten selaimen konsoliin tulostuu ”Mansikka”, ”Vadelma”, ”Loppu”. Koodissa on lisäksi siirrytty käyttämään ECMAScript 6:n uutta syntaksia funktioliteraaleille, jossa funktion parametrit ovat nuolen vasemmalla ja sen runko nuolen oikealla puolella. Jatkoesimerkeissä käytetään tätä syntaksia sen lyhyyden ja selkeyden vuoksi.

4.3 Operaattorit

Operaattorit ovat funktioita, jotka luovat Observablesta uuden Observablen. Uusi Observable on usein jollakin tapaa muokattu, suodatettu tai koottu tapahtumavirta. Vanha Observable pysyy muuttumattomana ja siihen voidaan käyttää useita operaattoreita luomaan useampia uusia Observableja. RxJs 5 sisältää noin 100 valmista operaattoria ja niitä on myös mahdollista luoda lisää. Tässä esitellään niistä muutama tärkein.

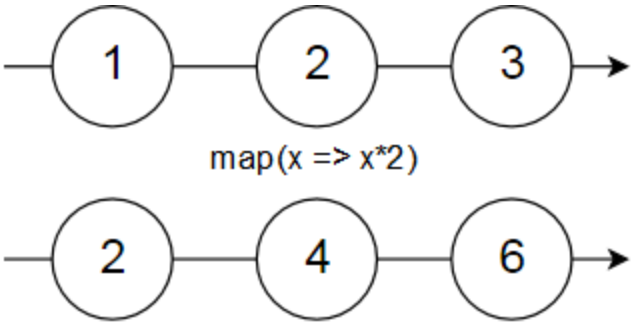
```
var hiiri = Rx.Observable.fromEvent(document, "mousemove");
var parilliset = hiiri.filter(e => e.clientX % 2 === 0 && e.clientY % 2 === 0);
parilliset.subscribe(e => console.log(e.clientX + ", " + e.clientY));
```

Esimerkkikoodi 28. Observablen luominen hiiren kursorin koordinaateista ja sen suodattaminen operaattorilla.

Esimerkkikoodissa 28 on luotu Observable hiiren kursorin koordinaateista selainikkunassa laskettuna pikseleissä vasemmasta yläkulmasta. Se emittoi koordinaatit sisältävän tapahtuman joka kerta, kun kursorin sijainti muuttuu. Tämän jälkeen siihen on käytetty filter-operaattoria luomaan uusi Observable. Filter ottaa vastaan funktion, joka palauttaa totuusarvon. Tässä tapauksessa filterille on annettu funktio, joka palauttaa

toden, mikäli molemmat koordinaatit ovat parillisia. Pelkät parilliset koordinaatit sisältävä Observable tilataan funktiolla, joka tulostaa ne muodossa ”100, 200” selaimen konsoliin.

Map on operaattori, jolla suoritetaan jokin muunnos virran jokaiselle elementille. Se ottaa vastaan funktion, jolla yksittäinen elementti muutetaan.



Kuva 11. Map-operaattorin käyttö.

Kuvassa 11 map-operaattoria on käytetty kokonaislukuja sisältävään virtaan. Sille on annettu funktio, joka kaksinkertaistaa vastaanotetun argumenttinsa. Näin luodussa uudessa virrassa on ensimmäisen elementit kaksinkertaistettuna. Uusi virta päivittyy sitä mukaa, kun ensimmäinen emittoi jotain, sillä se riippuu siitä.

Observableja voidaan yhdistää merge-operaattorilla.

```
var sekunti = Rx.Observable.interval(1000).map(e => "Sekunti kulunut");
var puoli = Rx.Observable.interval(500).map(e => "Puoli sekuntia kulunut");
sekunti.merge(puoli).subscribe(console.log);
```

Esimerkkikoodi 29. Merge-operaattorin käyttö.

Esimerkkikoodissa 29 luodaan kaksi Observablea interval-funktiolla, joihin emittoidaan tapahtuma annetun aikamäärän välein millisekunneissa. Ensimmäiseen tulee tapahtuma 1000 ms ja toiseen 500 ms välein. Niihin käytetään map-operaattoria muuttamaan tapahtumat kullekin omallensa merkkijonoksi, jotta on selvää, kummasta virrasta tapahtuma on peräisin. Operaattorit voi ketjuttaa toisiinsa kuten map ketjutetaan tässä suoraan Observableen, eikä niitä ole tarpeen sijoittaa välimuuttujiin. Sekunnin intervallivirtaan käytetään merge-operaattoria argumenttina puolen sekunnin virta ja

tulos tilataan console.log-funktiolla. Ei ole väliä, missä järjestyksessä kaksi virtaa yhdistetään. Lopputuloksena selaimen konsoliin tulostuu puolen sekunnin välein ”Puoli sekuntia kulunut” ja sekunnin välein ”Sekunti kulunut”.

FlatMap on aiempia esimerkkejä monimutkaisempi operaattori. Se ottaa vastaan funktion, joka kuten mapissa emittoi elementtikohtaisesti uuden elementin, mutta flatMapissa nämä emittoidut elementit ovat Observableja, jotka voivat sisältää elementtejä. FlatMap yhdistää mergen tavoin tulosvirtaan kaikkien näiden tämän funktion tuottamien virtojen elementit. Se ei siis tuota virtaa, jota kokoelmista puhuttaessa vastaisi sisäkkäiset taulukot, vaan kaikkien tulosvirtojen elementit koottuna.

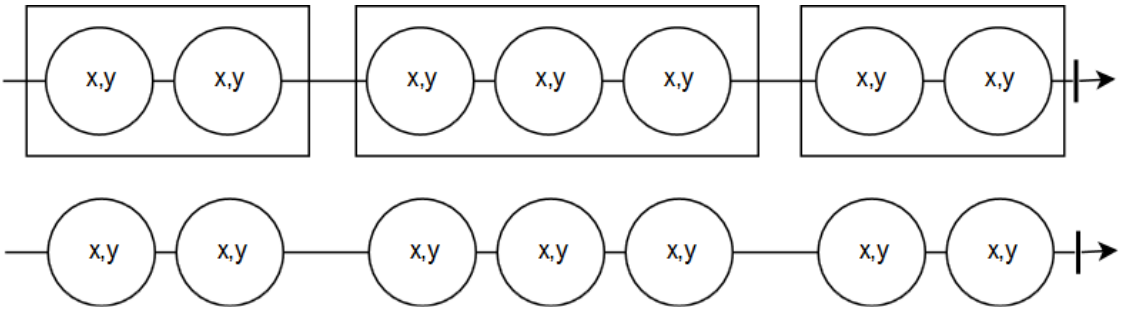
```
var laatikko = document.getElementById("laatikko");
var nappiAlas = Rx.Observable.fromEvent(laatikko, "mousedown");
var nappiYlos = Rx.Observable.fromEvent(laatikko, "mouseup");
var hiiri = Rx.Observable.fromEvent(document, "mousemove");
var siirto = nappiAlas.flatMap(klikkaus => {
  return hiiri.map(kursori => {
      return {
        vasenOffset: kursori.clientX,
        ylaOffset: kursori.clientY
      };
    }).takeUntil(nappiYlos);
});

siirto.subscribe(e => {
  laatikko.style.top = e.ylaOffset + "px";
  laatikko.style.left = e.vasenOffset + "px";
});
```

Esimerkkikoodi 30. FlatMap-operaattorin käyttö.

Esimerkkikoodissa 30 on tapaus, jossa flatMapin käyttö on tarpeellista. Siinä toteutetaan RxJs:llä yksinkertainen laatikon hiirellä raahaaminen (drag and drop), jossa hiiren nappi painetaan alas laatikon kohdalla ja sitä siirretään liikuttamalla hiirtä. Laatikko pysähtyy, kun hiiren nappi nostetaan. Esimerkissä tarvitaan kolmea virtaa: kursorin liikkeitä ikkunassa sekä hiiren napin painamista ja ylöspäästöä laatikon kohdalla. Observable,

joka on nimetty siirroksi emittoi kursorin koordinaatteja hiiren napin painamisesta sen ylöspäästämiseen asti. Virta lopetetaan takeUntil-operaattorilla, joka kuuntelee nappiYlos-virtaa.

FlatMap on tarpeellinen esimerkissä juuri virran loppumisen takia – laatikkoa tulee voida raahata useita kertoja. Mikäli virtaa kuunneltaisiin vain kerran, takeUntil poistaisi siltä kuuntelijat lopettaessaan sen. Siirrot-virran voidaan ajatella kapseloivan useita virtoja kokonaisista siirroista, yhdistäen ne kuitenkin sisältämään yksinkertaisia elementtejä eikä kokonaisia Observableja.



Kuva 12. flatMap-operaattorin tuottamat virrat vaiheittain.

Kuvassa 12 on marble-diagrammi flatMapin tässä esimerkissä tuottamasta virrasta. Ylemmässä virrassa suorakulmioilla on ilmaistu flatMapin ensimmäistä vaihetta, jossa virta sisältää virtoja. FlatMap ei kuitenkaan palauta tätä virtaa, vaan poimii vain virtojen sisältämät elementit lopputulokseen. Elementtiryppäät alemmassa, flatMapin palauttamassa virrassa tarkoittavat koordinaattipareja yhdelle siirrolle alusta loppuun ja välit aikoja, jolloin hiiren nappi oli ylhäällä eikä virtaan tullut uusia elementtejä.

5 Yhteenveto

Tässä opinnäytetyössä kuvattiin reaktiivinen ohjelmointi ohjelmoinniksi asynkronisilla datavirroilla ja vähennettiin epämääräisyyttä sen eri määritelmissä. Sen todettiin soveltuvan hyvin frontendin web-ohjelmointiin, jonka monimutkaistuessa haasteena juuri on useista lähteistä saapuvien asynkronisten tapahtumien käsittely ja hallinta. RxJs-esimerkeillä pyrittiin antamaan kattava alkupiste DOM-tapahtumien ja Ajax-kutsujen

hallitsemiseen reaktiivisesti. Elm-esimerkillä todettiin lisäksi, että frontend-ohjelmointia voi myös harrastaa JavaScriptistä merkittävästi eroavilla ohjelmointikielillä.

Selain ja JavaScript ohjelmointiympäristönä ovat monimuotoinen ja työkaluiltaan nopeasti muuttuva kokonaisuus, jossa parhaat toimintatavat vaihtuvat nopeasti. Sen erityishaasteiden ymmärtämiseksi erityisesti asynkronisuuteen liittyen oli tarpeellista käydä läpi HTML:n ja JavaScript-kehityksen perusteet, jotta voidaan tyydyttävästi kuvata, miksi sen tiettyihin ongelmiin on haettu ratkaisua paradigmarajojen yli.

Uusien ajattelumallien omaksuminen vaatii aina aikaa ja vaivaa. Voi olla vaikea arvioida, ratkaiseeko niillä todella jonkin ongelman vai ovatko ne pelkkiä trendejä. Johtopäätöksenä reaktiivisen ohjelmoinnin saralta voidaan todeta, että se on elegantimpi ja yksinkertaisempi tapa hallita tapahtumapohjaista ohjelmointia perinteisten tapahtumankäsittelijöiden sijaan.

Vaikka jokaista yhden lomakkeen sisältävää sivua ei välttämättä ole tarpeen abstraktoida asynkroniseksi tapahtumavirraksi, reaktiivinen ohjelmointi on kuitenkin hyvä työkalu, johon perehtyä asynkronisuuden lähteiden kasautuessa. Web-sovelluksien monimutkaistuminen jatkuu, ja uskon, että reaktiivinen ohjelmointi ja sen periaatteet tulevat nousemaan entistä keskeisimmiksi.

Lähteet

1 Fink, G. & Flatow, I. 2014. Pro Single Page Application Development: Using Backbone.js and ASP.NET 1st Edition. Apress.

2 Mansilla, S. 2015. Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code. Pragmatic Bookshelf.

3 Reactive Forms. <https://angular.io/docs/ts/latest/guide/reactive-forms.html>. Luettu 4.4.2017.

4 Reactive programming vs. Reactive systems. <https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems> Luettu 4.4.2017.

5 Christensen, B. & Nurkiewicz, T. 2016. Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications. O'Reilly Media.

6 The introduction to Reactive Programming you've been missing. <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Luettu 5.4.2017.

7 Blackheath, S. & Jones, A. 2016. Functional Reactive Programming. Manning Publications.

8 Davies, A. 2012. Async in C# 5.0: Unleash the Power of Async. O'Reilly Media.

9 Elliott, C. & Hudak, P. 1997. Functional Reactive Animation. International Conference on Functional Programming.

10 What is (functional) reactive programming? <http://stackoverflow.com/a/1030631>. Luettu 5.4.2017.

11 Abelson, H. & Sussman G. 1996. Structure and Interpretation of Computer Programs - 2nd Edition. The MIT Press.

12 The Reactive Manifesto. <http://www.reactivemanifesto.org>. Luettu 6.4.2017.

13 Gamma, E. & Helm, R. & Johnson, R. & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. 1994. Addison-Wesley Professional.

14 ReactiveX <http://reactivex.io/>. Luettu 7.4.2017.

15 The History of the Web. <https://www.w3.org/wiki/The_history_of_the_Web>. Luettu 7.4.2017.

16 Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.

17 Crockford, D. 2008. JavaScript: The Good Parts. O'Reilly Media.

18 Details of the Object Model – JavaScript | MDN. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model>. Luettu 8.4.2017.

19 A Short History of JavaScript. <https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript>. Luettu 8.4.2017.

20 Speed test: Google Chrome beats Firefox, IE, Safari. <https://www.cnet.com/news/speed-test-google-chrome-beats-firefox-ie-safari/>. Luettu 8.4.2017.

21 ECMAScript 2017 (ES8): the final feature set. <http://2ality.com/2016/02/ecmascript-2017.html>. Luettu 8.4.2017.

22 Flanagan, D. 2011. JavaScript: The Definitive Guide: Activate Your Web Pages - 6th edition. O'Reilly Media.

23 Swartz, A. 2005. A Brief History of Ajax. <http://www.aaronsw.com/weblog/ajaxhistory>. Luettu 8.4.2017.

24 Beschastnikh, I. & Gallaba, K. & Mesbah, A. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.

25 Promise – JavaScript | MDN. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>. Luettu 8.4.2017.

26 Ashkenas, J. List of languages that compile to JS. <https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js>. Luettu 9.4.2017.

27 Closure Compiler | Google Developers. <https://developers.google.com/closure/compiler/>. Luettu 9.4.2017.

28 The WebSocket API. <https://www.w3.org/TR/websockets/>. Luettu 12.4.2017.

29 Thinking in React. <https://facebook.github.io/react/docs/thinking-in-react.html>. Luettu 9.4.2017.

30 Motivation – Redux. <http://redux.js.org/docs/introduction/Motivation.html>. Luettu 10.4.2017.

31 Czaplicki, E. 2012. Elm: Concurrent FRP for Functional GUIs. Harvard University.

32 Zukowski, J. 1997. Java AWT Reference. O'Reilly Media.

33 RxJs API Document. <http://reactivex.io/rxjs/>. Luettu 11.4.2017.

34 Wasson, M. 2013. ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>. Luettu 12.4.2017.

35 Saumont, P. 2014. What's Wrong with Java 8: Currying vs Closures. <https://dzone.com/articles/whats-wrong-java-8-currying-vs>. Luettu 18.4.2017.