TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HENRIK HARTIALA

ASYNKRONISUUS JAVASCRIPTISSÄ

Kandidaatintyö

Tarkastaja: Pia Niemelä

TIIVISTELMÄ

HENRIK HARTIALA: Asynkronisuus JavaScriptissä
Asynchrony in JavaScript
Tampereen teknillinen yliopisto
Kandidaatintyö, 20 sivua
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Pia Niemelä

Avainsanat: Asynkronisuus, JavaScript, promise, callback, async/await, web-kehitys

Tässä työssä käydään läpi web-kehityksen historian vaiheita synkronisista HTML-sivuista kohti asynkronista ohjelmointia. Asynkronisen web-ohjelmoinnin johdosta käyttäjän ei tarvitse odottaa toimeettomana esimerkiksi datahakua palvelimelta, vaan voi käyttää sovellusta samanaikaisesti. Työssä tarkastellaan eri tapoja toteuttaa asykronisuutta JavaScriptissä, näiden tapojen kehitysvaiheet, sekä niiden mukanaan tuomat ongelmat ja ratkaisut. Työ keskittyy eri toteutustapojen luettavuuteen, ymmärrettävyyteen ja muokattavuuteen. Lisäksi työssä käydään vielä läpi virheiden käsittelyä näissä toteutustavoissa.

Työssä toteutetaan sama demo-ohjelma kolmella eri asynkronisella tavalla JavaScriptissä ja vertaillaan näiden luettavuutta, muokattavuutta ja virheiden käsittelyä. Tämän lisäksi pureudutaan vielä *async/await*:in mukanaan tuomiin ongelmiin toisella demo-ohjelmalla.

Työssä todetaan, että asynkronisuuden toteutus callback-metodin avulla JavaScriptissä muuttuu erittäin vaikealukuiseksi ja vaikeasti muokattavaksi ohjelman monimutkaistuessa, sillä se ajaa koodin rakenteen pyramidimaiseksi sisäkkäisyydeksi. Tähän ongelmaan kehitetty promise toimii erinomaisena ratkaisuna lohkojakonsa ja ketjutettavuutensa ansiosta. Promisen lisäksi kehitetty *async/await*-metodi tuo vielä keinon kirjoittaa promisepohjaista asynkronista koodia ikään kuin se olisi synkronista, mutta työssä todetaankin, että tämän metodin kanssa täytyy olla tarkka missä ja miten sitä käytetään. Väärin käytettynä metodi vain hidastaa koodin suoritusta.

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
DOM	Document Object Model
AJAX	Asynchronous JavaScript And XML
XHR	XMLHttpRequest
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
XML	Extensible Markup Language
W3C	World Wide Web Consortium

1. JOHDANTO

Web-sovellusten ja työpöytäsovellusten ohjelmointi eroavat toisistaan paljon. Toisin kuin työpöytäympäristössä, selaimilla on tarjota yksi säie kaikille, jotka tarvitsevat pääsyä käyttöliittymään. Yksittäinen säikeistysmalli rajoittaa käyttöliittymän elementtejä muokkaavaa sovelluskoodia, koska se rajoittaa samalla muun koodin suorittamista. Toisin sanoen funktiot ja säikeet estävät käyttöliittymän käytön, kunnes säie on suoritettu. Tästä syystä on tärkeää hyödyntää kaikkia selaimen tarjoamia asynkronisia toimintoja.

Vielä 1990-luvun puolessa välissä suurin osa verkkosivuista koostui kokonaan synkronisista HTML-sivuista, jolloin jokainen käyttäjän toiminto vaati koko sivun lataamisen uudestaan palvelimelta [19]. Sivulla ei voinut tehdä muita toimintoja sillä välin, kun selain odotti palvelimelta dataa. Jälkeenpäin on kehitetty teknologioita, joiden avulla websovellukset pystyvät lähettämään ja vastaanottamaan dataa palvelimelta asynkronisesti niin, että sivua tai käyttöliittymää ei tarvitse ladata kokonaan uudelleen, eikä yksittäisen toiminnon odotus lukitse koko sivun käyttöä samalla. Tässä työssä käydään läpi webkehityksen historiaa, sekä perehdytään asynkronisuuden toteutustapoihin ja ongelmakohtiin JavaScriptissä.

Toisessa luvussa käydään läpi web-kehityksen pääkomponenttien historiaa, eli mitä webkehitys oli ennen, mihin on nyt päästy ja miten tähän päästiin. Kolmannessa luvussa tuodaan esille miten synkroninen ja asynkroninen ohjelmointi eroavat toisistaan, miten ne käytännössä toimii ja miten ne liittyvät web-kehitykseen. Neljännessä luvussa esitellään koodiesimerkein kolme tapaa toteuttaa asynkronisuutta JavaScriptissä. Viidennessä luvussa toteutetaan demo-ohjelmat, joiden avulla havainnollistetaan ongelmakohtia ja ratkaisuja eri tavoissa toteuttaa asynkronisuutta JavaScriptissä. Kuudennessa luvussa tehdään yhteenveto työstä.

2. WEB-KEHITYKSEN PÄÄKOMPONENTIT

Web-kehityksen pääkomponenteista ensimmäiset ovat *HyperText Markup Language* (HTML) ja *Casccading Style Sheets* (CSS) [17]. HTML määrittelee web-dokumentin rakenteen ja asettelun. Selaimet koostavat HTML:stä käyttäjälle näkyvän web-sivun ja CSS:ää voidaan käyttää HTML-sivun ulkoasun lisämäärittelyyn. Tämän tarkoitus on erotella itse HTML-sivun sisältö sen esitystavasta, johon kuuluu esimerkiksi väritykset ja fontit [5]. CSS:än avulla kehittäjät voivat sisällyttää useampia tyylisääntöjä samalle sivulle, jotta käyttötuntumaa saadaan muokattua erilaiseksi saman sivuston eri sivuille [4]. Håkon Wium Lie ehdotti CSS:ää ensimmäisen kerran vuonna 1994 HTML-tyyliohjeiden standardiksi W3C:lle (*World Wide Web Consortium*) [8], joka toimii web-standardien ylläpitäjänä ja kehittäjänä [24]. CSS:än ensimmäinen vaihe sai suosituksen W3C:lta vuonna 1998 [21].

Seuraava pääkomponenteista on *Document Object Model* (DOM), jota käytetään sivun dynaamiseen esitykseen ja vuorovaikutukseen sisällön kanssa. Ladatessa selaimeen web-dokumentti muunnetaan objekteiksi, joita voidaan helposti käsitellä toisistaan erillään [10]. DOM:in avulla saadaan lisättyä interaktiivisuutta web-dokumentteihin samalla tavalla, kuin CSS:än avulla tyylejä. Vuonna 1997 julkaistiin selaimet *Netscape Navigator 4* ja *Internet Explorer 4*, joihin luvattiin monia lisäyksiä mitä JavaScript:illä voitaisiin tehdä laajentuneen DOM:in avulla. Kehittäjiä rohkaistiin testaamaan uutta *Dynamic HTML*:ää (DHTML), joka siis koostui HTML:stä, CSS:stä ja JavaScriptistä. Netscapen ja Microsoftin selaimilla oli kuitenkin keskenään täysin erilainen lähestymistapa DOM:iin ja näin ollen olivat keskenään yhteensopimattomia. Samoihin aikoihin W3C oli alkanut koota standardoitua DOM:ia, jonka kehitykseen Microsoft ja Netscape lopulta liittyivät. Tulokseksi saatu ensimmäisen vaiheen standardoitu DOM sai W3C-suosituksen vuonna 1998 [22].

Vuonna 1999 *Internet Explorer 5.0*:n mukana esiteltiin *XMLHTTPRequest -objekti* (XHR), jonka avulla verkkosivut voivat lähettää ja vastaanottaa dataa HTTP-protokollan kautta [20]. Tämä mahdollisti sen, ettei koko verkkosivua tarvinnut aina ladata uusiksi näyttääkseen palvelimelta saadun datan. Vuonna 2005 Jesse James Garrett esitteli uuden termin AJAX, eli *Asynchronous JavaScript + XML* (Extensible Markup Language) [9]. AJAX sisällyttää HTML:än, CSS:än, DOM:in, XML:än, asynkronisen datahaun käyttäen XHR:ää ja JavaScriptin millä kaikki edellä mainitut saadaan sulavasti liitettyä yhteen. Tähän aikaan klassinen web-sovellus-malli toimi niin, että käyttäjän toimista lähetettiin HTTP-pyyntö palvelimelle, missä palvelin käsitteli pyyntöä, teki tarvittavia toimenpiteitä ja laskuja, kunnes lopulta palautti vastauksena HTML-sivun käyttäjälle. Kaiken tämän aikaa käyttäjä joutui odottamaan, eikä tämä ollut tietenkään käyttäjäystävällistä. AJAX-ohjelmointi muutti tämän kokonaan, sillä istunnon alussa selain lataa web-sivun sijaan AJAX-moottorin, joka on kirjoitettu JavaScriptillä. AJAX-moottori vastaa

näkymän päivittämisestä käyttäjälle, sekä kommunikoinnista palvelimen kanssa. Tämän avulla käyttäjän toiminnasta aiheutuvat palvelinpyynnöt tapahtuvat asynkronisesti, jolloin web-sovelluksen käyttö ei esty datahaun ajaksi.

XML on perinteinen datansiirtoformaatti AJAX:issa, jonka etuina ovat standardisointi ja skaalautuvuus. XML:n kehitys aloitettiin vuonna 1995 ja saavutti W3C-standardoinnin vuonna 1998 [11]. XML:ssä data esitetään puurakenteen muodossa; elementteinä ja attribuutteina.

```
<users>
  <user>
    <id>1</id>
    <name>Maija</name>
  </user>
  <user>
    <id>2</id>
    <name>Matti</name>
  </user>
</users>
```

Listaus 1: XML -esimerkki

Listauksen 1 esimerkissä oleva *users* sisältää kaksi instanssia luokasta *user*, jotka sisältävät kaksi attribuuttia: *id* ja *name*.

XML:n rinnalle kehitetty *JavaScript Object Notation* (JSON) on XML:stä kevyempi datansiirtoformaatti. JSON:ia on helppo lukea ja kirjoittaa, ja toisin kuin XML:ssä, sen generoiminen ja parsiminen on nopeaa. JSON:issa data muodostuu arvojoukoista muodostuvista taulukoista, ja objekteista jotka muodostuvat nimi/arvo -pareista.

```
[
  {
    "id": "1",
    "name": "Maija"
  },
  {
    "id": "2",
    "name": "Matti"
  }
]
```

Listaus 2: JSON -esimerkki

Listauksessa 2 on listauksen 1 esimerkki yksinkertaisimmillaan JSON -formaatissa. Hakasulkeet määrittävät taulukon, jonka sisältämät objektit erotellaan pilkulla. Nimi/arvo -parit ovat heittomerkeissä kaksoispisteellä eroteltuna ja jokainen pari on eroteltu pilkulla.

3. SYNKRONISUUS JA ASYNKRONISUUS

Ohjelmoinnissa puhuttaessa synkroninen suoritus tarkoittaa sitä, että jokainen koodirivi suoritetaan järjestyksessä, eikä seuraavan rivin suoritus voi alkaa ennen kuin aiempi rivi on suoritettu [3]. Funktiokutsuissa ohjelman suoritus odottaa, että funktiosta palataan, jotta voidaan jatkaa suoritusta seuraavalta riviltä. Jos em. funktiossa tehdään aikaa vieviä toimintoja, kuten datan noutamista palvelimelta, pääohjelma odottaa koko tämän ajan ennen kuin jatkaa oman koodin suoritusta. Asynkronisessa suorituksessa funktiosta palaamista ei jäädä odottamaan, vaan jatketaan välittömästi pääohjelman suoritusta.

```
1  function threeSecondFunction() {
2    setTimeout(() => {
3      console.log('One');
4    }, 3000)
5  }
6  function anotherThreeSecondFunction() {
7    setTimeout(() => {
8      console.log('Two');
9    }, 3000)
10 }
11 threeSecondFunction();
12 anotherThreeSecondFunction();
13 console.log('Three');
```

Listaus 3: Esimerkki: Synkroninen ja asynkroninen suoritus

Listauksessa 3 olevat *threeSecondFunction* ja *anotherThreeSecondFunction* ovat siis funktioita, joissa *setTimeoutin* sisäiset rivit(3, 8) suoritetaan *3000 ms* jälkeen. Nyt jos ohjelmaa suoritetaan rivien 11–13 mukaisesti synkronisesti, kestää ensin kolme sekuntia ja ohjelma tulostaa "One", kuluu kolme sekuntia lisää ja ohjelma tulostaa "Two"ja heti perään "Three". Asynkronisessa suorituksessa ohjelma tulostaa ensin "Three", kuluu kolme sekuntia ja ohjelma tulostaa peräkkäin "One"ja "Two".

Web-sovelluksen laajetessa ja monimutkaistuessa sovellus tekee yhä useampia datahakuja yleensä eri palvelimilta, jolloin synkronisessa suorituksessa jokaista datahakua jouduttaisiin odottamaan erikseen ja käyttäjäkokemus kärsisi [12]. Asynkronisuuden avulla voidaan käyttää web-sovellusta normaalisti myös sen aikana, kun dataa haetaan palvelimilta. Kun sovellus lopulta saa vastauksen palvelimelta, voidaan tämä käsitellä esimerkiksi callback-metodilla, josta lisää kappaleessa 4.

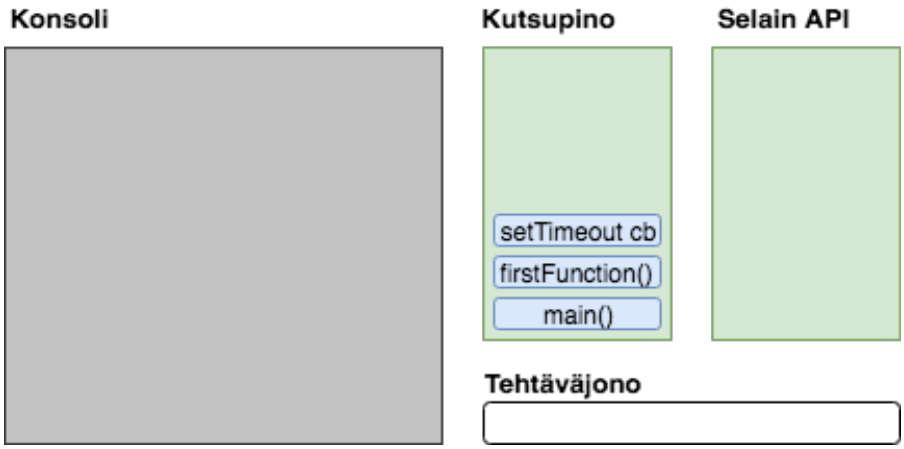
JavaScript käyttää ohjelman suorituksessa tapahtumapohjaista mallia, jolla on yksi suoritussäie [6]. Tässä mallissa *event loop* käsittelee tehtäväjonon tehtäviä yksi kerrallaan, luo-

den jokaiselle suorituksessa olevalle tehtävälle kutsupinon. Vasta kun kutsupino on suoritettu kokonaan, voidaan tehtäväjonosta ottaa uusi tehtävä käsittelyyn. Kun kutsupinossa suoritetaan asynkronista toimintoa, ne siirtyvät selaimen API:lle joko ajastettavaksi kuten *setTimeout*:in tapauksessa, tai palvelinkutsujen tapauksessa odottamaan vastausta palvelimelta. Sieltä ne siirretään tehtäväjonoon kun kyseinen operaatio on suoritettu. Seuraava esimerkki havainnollistaa tapahtumapohjaista mallia JavaScriptissä. [2]

```
1  function main() {
2    function firstFunction() {
3      setTimeout(function cb() {
4        secondFunction();
5      }, 0}
6    }
7    function secondFunction() {
8      console.log('Im second function');
9    }
10
11   firstFunction();
12   console.log('Done');
13 }
14 main();
```

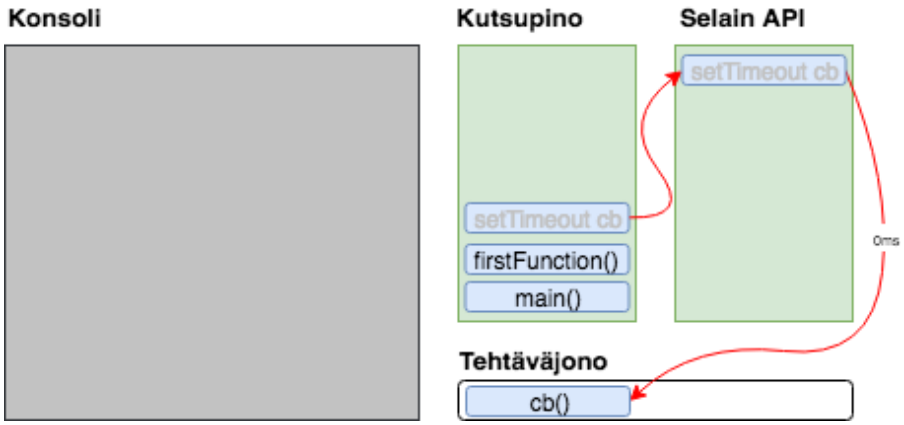
Listaus 4: Event loop -esimerkin koodi

Kutsupino ja tehtäväjono ovat aluksi tyhjiä. Kun listauksen 4 mukaista koodia suoritetaan, menee kutsupinoon ensin funktiokutsu *main()* riviltä 14. Tämän jälkeen kutsupinoon menevät funktiokutsu *firstFunction()* ja sen sisältä *setTimeout cb* kuvan 1 mukaisesti.



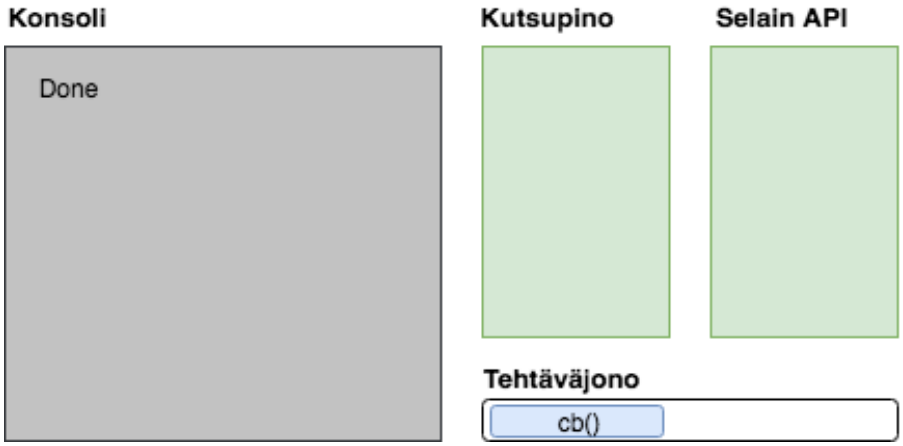
Kuva 1. *Event loop -esimerkin vaihe 1*

Suorittaessa *setTimeout*-kutsua, siirretään se kuvan 2 mukaisesti asykronisena toimintona selaimen API:lle ajastuksen ajaksi, josta sen callback-funktio *cb* siirtyy tehtäväjonon perälle kun aika on kulunut; tässä tapauksessa annettu aika 0, joten se siirtyy jonon perälle heti. Tehtäväjonossa seuraavana oleva tehtävä voidaan ottaa käsittelyyn vasta sen jälkeen, kun suorituskessa oleva kutsupino on saatu käsiteltyä.



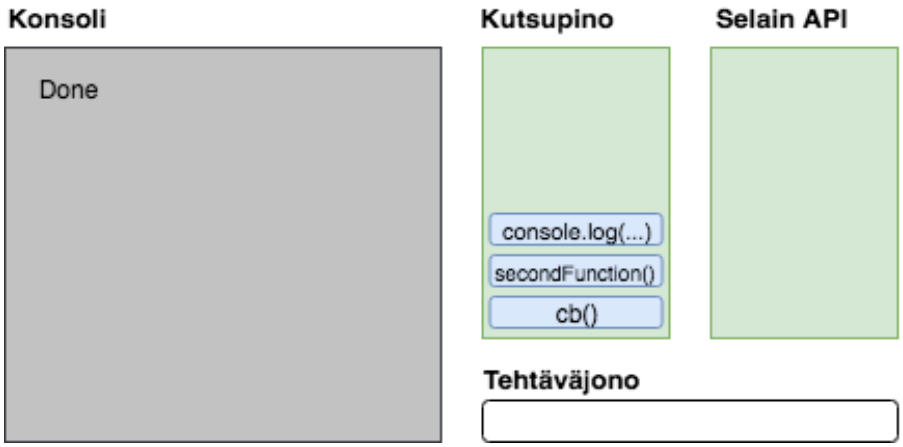
Kuva 2. *Event loop -esimerkin vaihe 2*

Kun *setTimeout cb* on poistettu kutsupinosta, voidaan siitä poistaa *firstFunction()*, koska funktio on nyt suoritettu. Seuraavaksi kutsupinoon menee listauksen 4 rivi 12 joka suoritetaan heti, konsoliin tulostuu *Done* ja voidaan poistaa se kutsupinosta. Funktio *main()* on suoritettu ja sekin voidaan poistaa kutsupinosta ja päästään kuvan 3 mukaiseen tilaan.



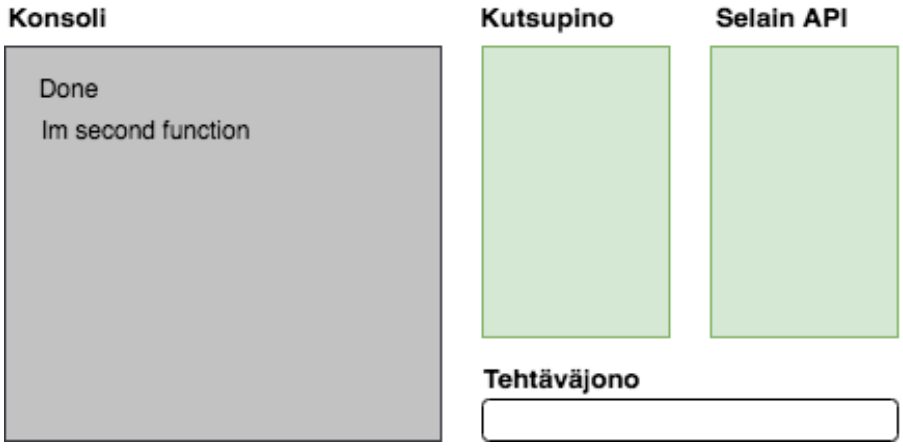
Kuva 3. *Event loop -esimerkin vaihe 3*

Suorituksessa ollut kutsupino on suoritettu, joten tehtäväjonosta otetaan seuraava tehtävä suoritukseen. Kutsupinoon tulee siis järjestyksessä *cb()*, *secondFunction()* ja *console.log('Im second Function')* kuvan 4 mukaisesti.



Kuva 4. *Event loop -esimerkin vaihe 4*

Lopuksi suoritetaan *console.log* joka tulostaa konsoliin *Im second function* ja kutsupinosta voidaan poistaa siellä olevat päältäpäin yksi kerrallaan, sillä ne ovat suoritettu. Näin päästään kuvan 5 mukaiseen lopputilanteeseen.



Kuva 5. *Event loop -esimerkin vaihe 5*

Kuten esimerkistä voidaan myös päätellä, niin *setTimeout*-metodi ei siis takaa sitä, että sen *callback*-funktio suoritettaisiin täsmälleen parametrina annetun ajan päästä, vaan tämä funktio siirretään annetun ajan kuluttua tehtäväjonon perälle.

4. ASYNKRONISUUDEN TOTEUTUS JAVASCRIPTISSÄ

4.1 Callback

Callback-funktioksi kutsutaan sellaista funktiota, joka annetaan parametrina toiselle funktiolle. Callback-funktio suoritetaan joko heti ennen kuin funktio palaa tai vasta sen jälkeen kun funktiosta ollaan jo palattu [6]. Jos callback-funktio suoritetaan heti, on kyseessä synkroninen callback, kuten esimerkiksi taulukon *forEach*-metodi missä taulukon jokaiselle alkiolle suoritetaan sama koodilohko [13]. Asynkroninen callback-funktio suoritetaan tulevaisuudessa, kuten esimerkiksi silloin kun HTTP-pyyntö saa vastauksen palvelimelta.

```
function getUsernameById(userId, callback) {
  //GET USERNAME THAT MATCHES ID FROM SERVER
  var userName = ...
  //OR REASON FOR ERROR
  var error = ...
  //
  if(userName){
    callback(null, userName);
  }else{
    callback(error, null);
  }
}

function getNames(userIdList) {
  userIdList.forEach(userId => {
    getUsernameById(userId, function(error, result) {
      if(!error) {
        console.log(result);
      }else{
        console.log(error);
      }
    });
  });
}
```

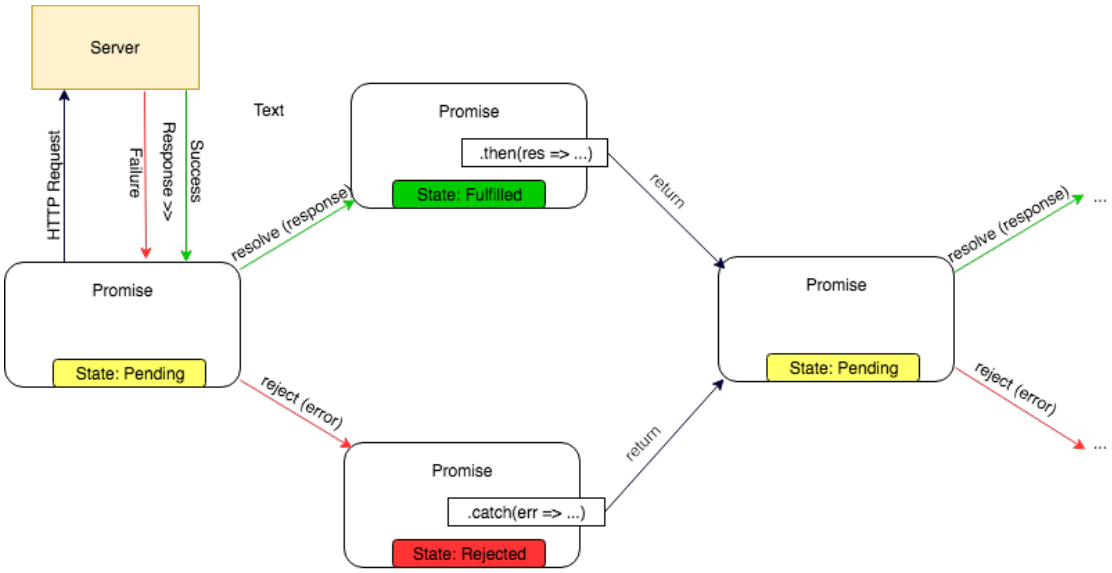
Listaus 5: Callback-funktiot

Listauksessa 5 rivillä 1 oleva funktio *getUsernameById* ottaa parametreikseen käyttäjän Id:n ja callback-funktion. Tämän jälkeen haetaan tietokannasta (tapaa ei määritelty) id:tä vastaava käyttäjätunnus ja sen löytyessä kutsutaan callback-funktiota antaen parametriksi tämä käyttäjätunnus. Rivillä 14 oleva funktio *getNames* ottaa parametreikseen listan, jossa on käyttäjien id:t. Funktio kutsuu *forEach*-metodilla jokaiselle listan id:lle *getUsernameById* funktiota, antaen tälle vuorossa olevan id:n sekä callback-funktion, jota kutsutaan kun käyttäjänimi on saatu selville. Tässä esimerkissä käyttäjätunnus tulostetaan. JavaScriptin sisäänrakennettu *try/catch*-metodi virheiden käsittelyyn ei asynkronisten callbackien kanssa toimi, joten kehittäjät ovat todenneet parhaaksi käytännöksi error first -protokollan [6]. Tämän käytännön mukaan varataan callbackin ensimmäinen parametri mahdolliselle virheelle, kuten listauksen 5 rivillä 16.

JavaScriptin callback-funktiot voivat olla nimettyjä tai nimettömiä funktioita [6]. Nimettyä callback-funktiota voi kutsua myös muualta ja nimetyt callback-funktiot auttavat debuggauksessa, sillä kutsupinoon jää funktion nimi. Nimetyt callback-funktiot kuitenkin pysyvät muistissa, eikä poistu roskien keruun yhteydessä. Nimettömät callback-funktiot eivät ole uudelleen käytettävissä, niitä on vaikeampi ylläpitää, debugata ja testata, mutta ne ovat resurssiystävällisempiä koska ne merkataan poistettavaksi muistista heti suorituksen jälkeen.

4.2 Promise

Promise-objekti edustaa arvoa mikä ei välttämättä ole vielä sillä hetkellä saatavilla, mutta tulee olemaan myöhemmin. Promise-objekti saa yleensä arvonsa asynkronisen operaation tuloksena. Promisella on kolme tilaa joissa se voi olla: pending, fulfilled ja rejected [14]. Pending-tilassa promisen arvoa ei ole vielä määritelty ja sen tila voi siirtyä joko fulfilled- tai rejected-tilaan. Promise siirtyy fulfilled-tilaan saatuaan onnistuneesti arvon ja rejected-tilaan, jos operaatio epäonnistui. Fulfilled-tilassa promisella on siis arvo ja rejected-tilassa promisella on syy epäonnistumiselle. Promise voi siirtyä pending-tilasta fulfilled- tai rejected-tilaan, mutta ollessaan fulfilled- tai rejected-tilassa sen arvoa tai tilaa ei voi enää muuttaa. Promisen *.then()*- ja *.catch()*-metodit palauttavat uuden promisen, jolloin näitä metodeja on mahdollista ketjuttaa useampia peräkkäin.



***Kuva 6**. Promisen tilasiirtymäkaavio*

Kuva 6 havainnollistaa promisen tilasiirtymiä. Kuvassa promise luodaan ja se suorittaa HTTP -pyynnön palvelimelle. Onnistuessaan promise siirtyy Fulfilled tilaan suorittamaan *.then()*-lohkoa, joka saa parametrikseen palvelimelta vastaukseksi saadun datan. Vaihtoehtoisesti jos datan hakemisessa tapahtuu virhe, niin promise siirtyy *Rejected*-tilaan käsittelemään virhettä *.catch()*-lohkossa. Nämä lohkot palauttavat uuden promisen, jolle voidaan toteuttaa uusi käsittelijä.

```
1  function getUsernameById(userId) {
2    return new Promise((resolve, reject) => {
3      //GET USERNAME THAT MATCHES ID FROM SERVER
4      var userName = ...
5      //OR REASON FOR ERROR
6      var error = ...
7      //
8      if(userName) {
9        resolve(userName);
10     }else{
11       reject(error);
12     }
13   )}
14 }
15
16 function getNames(userIdList) {
17   userIdList.forEach(userId => {
18     getUsernameById(userName)
19       .then(response => {
20         console.log(response);
21       })
22       .catch(error => {
23         console.log(error);
24       })
25   });
26 }
```

Listaus 6: Promise

Listauksen 6 funktio *getUsernameById* palauttaa nyt promise-objektin. Tämä objekti on tilassa pending, kunnes rivin 4 operaatio hakee sille arvon ja palauttaa sen rivillä 7 (fulfilled-tila), tai epäonnistumisen jälkeen palauttaa annetun virheen (rejected-tila) rivillä 11. *Resolve* ja *reject* ovat siis callback-funktiota, joita kutsutaan riippuen operaation onnistumisesta. Se miten toimitaan näitä callback-funktioita kutsuessa, määritellään ketjutettavien .then()- ja .catch()-metodien avulla. Operaation onnistuessa ja kutsuessa *resolvea* suoritetaan koodi .then()-lohkossa ja vastaavasti *rejectiä* kutsuessa suoritetaan koodi .catch()-lohkossa. Näitä callback-funktioita kutsuessa välitetään parametrina tarvittavat tiedot, kuten tässä rivillä 7 *userName* ja rivillä 11 *error*. Nämä parametrit vastaavat siis rivin 19 *responsea* ja rivin 22 *erroria*.

Promisen ketjutusta voidaan myös haarauttaa, eli promiselle voidaan kutsua useaa toisistaan erillistä *.then()*-metodia, kuten seuraavassa esimerkissä:

```
somePromise.then();
somePromise.then();
```

Tämä on siis eri asia, kuin:

```
somePromise.then().then();
```

Erona näillä kahdella on se, että ensimmäisessä esimerkissä sama promise on haarautettu kahteen erilliseen suorituslohkoon. Jälkimmäinen on ketjutettu ja saattaa toisessa *.then()* -lohkossa käsitellä eri promisea. [19]

Promise.all() on metodi joka saa arvonsa, kun sille parametriksi annetun listan kaikki promiset ovat saaneet arvonsa. Tuloksena saatu arvo on taulukko, joka sisältää jokaisen listassa olleen promisen arvon samassa järjestyksessä, kun missä ne annettiin. Jos jokin promise hylätään, niin *Promise.all()* saa arvoksi ensimmäisen hylätyn promisen virheen. [15]

```
Promise.all([promise1, promise2, promise3])
  .then(values => console.log(values))
  .catch(err => console.log(err));
```

Yllä olevassa esimerkissä metodille on anettu parametreiksi *promise1, promise2* ja *promise3*. Kun jokainen näistä promiseista on saanut arvonsa, saa *Promise.all()* myös arvonsa ja siirtyy *.then()*-lohkoon, missä *values* sisältää taulukon em. promiseiden arvoista. Jos yksinkin promise hylätään, siirrytään *.catch()* -lohkoon hylätyn promisen virheen kanssa. Tällä metodilla voidaan suorittaa jokin haluttu toiminto vasta sen jälkeen, kun kaikki halutut promiset ovat saaneet arvonsa.

Promise.race() on metodi [16], joka saa arvonsa kun sille parametriksi annetusta promiselistasta ensimmäinen saa arvonsa. Tämän metodin avulla voidaan luoda esimerkiksi aikakatkaisu siten, että annetaan parametreiksi haluttu promise ja tämän lisäksi toinen promise joka tietyn ajan päästä siirtyy *rejected*-tilaan ja palauttaa virheen. Näin ollen jos haluttu promise ei ole saanut arvoa tietyssä ajassa, voidaan toiminto katkaista ja antaa käyttäjälle virheilmoitus.

4.3 Async/Await

Async/await on tapa kirjoittaa promise-pohjaista koodia ikään kuin se olisi synkronista, mutta estämättä pääohjelman suoritusta [7]. Tämä saadaan mahdollistettua kirjoittamalla funktion määrittelyn eteen avainsana *async*, kuten listauksen 7 ensimmäisellä rivillä. Funktion sisällä voidaan nyt käyttää *await*-metodia. Odottaessa promisea voidaan antaa

avainsana *await*, jolloin funktion suoritus pysähtyy tähän estämättä pääohjelman suoritusta kunnes promise on ratkaistu. Kun odotetun promisen arvo on saatu, jatketaan funktion suorittamista. Jos promise hylätään (*reject*), heitetään poikkeus, joka voidaan käsitellä *.catch()*-lohkossa. *Async*-funktio palauttaa myös itse promisen, eli sille voidaan kutsua halutessaan .then()-metodia.

```
1  async function getNames(userIdList) {
2    userIdList.forEach(userId => {
3      try {
4        let username = await getName(userId);
5        console.log(username)
6      }
7      catch(err) {
8        console.log(err);
9      }
10   })
11 }
```

Listaus 7: Async/await

Listauksen 7 *getNames*-funktion edessä oleva avainsana *async* mahdollistaa *await*-metodin käytön. Nyt koko funktio pysähtyy odottamaan rivin 4 promisen arvoa tai heitettyä poikkeusta. Poikkeus käsitellään *.catch()*-lohkossa rivillä 7. Jokainen rivin 4 funktiokutsu tehdään yksi kerrallaan, koska *forEach* on myös *async*-funktion sisällä. Jos *async*-funktion alue kattaisi vain rivin 3 ja 9 välisen alueen, niin *forEach*-loopin kierrokset suoritettaisiin asynkronisesti, mutta jokainen yksittäinen kierros synkronisesti.

5. VERTAILU DEMO-OHJELMAT

Tässä kappaleessa toteutetaan sama sovellus käyttäen edellisessä kappaleessa esiteltyjä callback-funktioita, promiseja sekä *async/await*-metodia. Sovellukselle annetaan kolme merkkijonoa erikseen, jotka se yhdistää järjestyksessä ja lopulta tulostaa yhdistetyn merkkijonon. Jokainen yhdistäminen vie satunnaisen määrän aikaa nollan ja yhden sekunnin väliltä. Tässä esimerkkisovelluksessa lopputulostus on siis: *"First Second Third Fourth"*.

```
1  function combineStrings(previous, current, callback) {
2    setTimeout(() => {
3      callback(null, previous + ' ' + current);
4    }, Math.floor(Math.random() * 1000) + 1);
5  }
6
7  combineStrings('First', 'Second', (err, result) => {
8    if(err) {
9      console.log(err);
10   }else {
11     combineStrings(result, 'Third', (err, result) => {
12       if(err) {
13         console.log(err);
14       }else {
15         combineStrings(result, 'Fourth', (err, result) => {
16           if (err) {
17           console.log(err);
18           }else {
19             console.log(result);
20           }
21         });
22       }
23     });
24   }
25 });
```

Listaus 8: Callback-demo

Listauksen 8 funktio *printToLog* ottaa parametreikseen callback-funktion sekä kaksi merkkijonoa. Funktio yhdistää merkkijonot välilyönnillä eroteltuna ja kutsuu yhdistetyllä merkkijonolla callback-funktiota 1–1000 millisekunnin jälkeen. Jotta merkkijonot saadaan yhdistettyä järjestyksessä peräjälkeen niin, että pääohjelma ei esty, täytyy callback-funktiot kirjoittaa sisäkkäin, kuten riveillä 7–25. Tällainen rakenne syntyy, kun ensimmäisen callbackin tulos täytyy antaa toiselle callbackille parametrina. Mitä monimutkaisempi sovellus on, sitä enemmän sisäkkäisyyttä tapahtuu callbackien kanssa. Tätä pyramidimaista rakennetta kutsutaankin *callback hell* -ongelmaksi [18]. Koodin suoritusjärjestys on mo-

nimutkainen ja koodia on vaikea lukea, muokata ja debugata [19]. Promise ratkaisee erinomaisesti tämän ongelman, sillä sisäkkäisyyden sijaan koodin voi ketjuttaa.

```
1    function combineStrings(previous, current) {
2      return new Promise((resolve, reject) => {
3        setTimeout(() => {
4          resolve(previous + ' ' + current);
5        }, Math.floor(Math.random() * 1000) + 1)
6      })
7    }
8    combineStrings('First', 'Second')
9      .then(res => combineStrings(res, 'Third'))
10     .then(res => combineStrings(res, 'Fourth'))
11     .then(res => console.log(res))
12     .catch(err => console.log(err));
```

Listaus 9: Promise-demo

Listauksen 9 funktio *printToLog* palauttaa promise-objektin, eikä ota parametrikseen callback-funktiota. Pyramidimaisen callback-rakenteen sijaan promisella toteutettu ketjutus (rivit 8–12) on huomattavasti helpompi lukea. Sovelluksen kulku on selkeä ja sitä on helppo muokata. Virhetilanteessa promise palautuu hylättynä, mikä käsitellään *.catch()*-lohkossa, eikä virhettä tarvitse välittää ketjun läpi, kuten callbackien sisäkkäisyyden kanssa. Poikkeuksia ei siis kuulu käsitellä *.then()*-lohkoissa synkronisesti *try-catch*-metodilla, sillä se aiheuttaisi kaksinkertaisen poikkeusten käsittelyn [23].

Promiseista vielä yksinkertaisempi ja helppolukuisempi versio ohjelmasta saadaan *async/await*:in avulla. Listauksen 10 funktion *combineFour()* edessä avainsana *async* mahdollistaa riveillä 10–12 käytetyn *await*-metodin käytön, jossa kyseisen rivin suoritus odotetaan loppuun ennen seuraavan rivin suoritusta.

```
1  function combineStrings(previous, current) {
2    return new Promise((resolve, reject) => {
3      setTimeout(() => {
4        resolve(previous + ' ' + current);
5      }, Math.floor(Math.random() * 1000) + 1)
6    })
7  }
8  async function combineAndPrint() {
9    try {
10     var comboString = await combineStrings('First', 'Second');
11     comboString = await combineStrings(comboString, 'Third');
12     comboString = await combineStrings(comboString, 'Fourth');
13     console.log(comboString);
14   }catch(err) {
15     console.log(err);
16   }
17 };
```

Listaus 10: Async/await-demo

Asynkronisesti toimiva sovellus saatiin kirjoitettua perinteisemmällä eli synkronisella tavalla. Ongelmaksi alussa voi tulla se, että koodia lukiessa ei huomaa niin helposti mitkä osuudet ovat asynkronisia, koska avainsana *await* ei ole niin esiinpistävä kuin *callback*- tai *.then()-metodi*.

Async/await:in kanssa muodostuu helposti ongelmia, jos asynkroniset operaatiot ovatkin toisistaan riippumattomia. Seuraavassa esimerkissä tilataan kaksi tuotetta eri kaupoista ja laitetaan toinen tuote käyttöön ja toinen tuote myyntiin kun ne saapuvat. Lopuksi tulostetaan ensimmäisen tuotteen myynnistä saatu rahamäärä.

```
1 async function orderUseAndSellItems() {
2   var item1 = await orderItemShop1();  //SYNC
3   var item2 = await orderItemShop2();  //SYNC
4   useItem(item2);  //ASYNC
5   var payment = await sellItem(item1); //SYNC
6   console.log('I got ' + payment + ' euros as payment.');
7 }
```

Listaus 11: Async/await-demo 2

Listauksessa 11 ohjelma toimii, mutta nyt *item2* tilaus joutuu odottamaan turhaan *item1* tilauksen suoritusta, vaikka nämä voitaisiin suorittaa rinnakkain. Tämän jälkeen myös *item1* myyntiin laittaminen rivillä 5 joutuu odottamaan että *item2* on tilattu. Koska async/await on promise-pohjainen, voimme tehdä asynkroniset funktiokutsut ilman *await* -avainsanaa ja suorittaa odotus vasta siellä missä niitä tarvitaan.

```
1 async function orderUseAndSellItems() {
2   var item1 = orderItemShop1();  //ASYNC
3   var item2 = orderItemShop2();  //ASYNC
4   useItem(await item2);          //SYNC
5   var payment = await sellItem(await item1);  //SYNC
6   console.log('I got ' + payment + ' euros as payment.');
7 }
```

Listaus 12: Async/await-demo 2 korjausyritys

Listauksessa 12 käytetään *await*-metodia vasta käyttövaiheessa, eli riveillä 4 ja 5, jolloin tilaukset riveillä 2 ja 3 saadaan suoritettua asynkronisesti rinnakkain. Sen lisäksi, että ohjelmasta tuli nyt vaikealukuinen, niin ei tiedetä kumpi tuotteista saapuu ensin. Jos *item1* saapuu ensin, joutuu se turhaan odottelemaan *item2*:sen saapumista rivillä 4. Tämän korjaamiseksi täytyy tehdä kummallekin parille omat funktiot listauksen 13 mukaisesti.

```
async function orderAndSell() {
  var item1 = await orderItemShop1();
  var payment = await sellItem(item1);
  console.log('I got ' + payment + ' euros as payment.');
}
async function orderAndUse() {
  var item2 = await orderItemShop2();
  useItem(item2);
}
orderAndSell();
orderAndUse();
```

Listaus 13: Async/await-demo 2 korjaus

Toisistaan riippuvat osuudet on nyt omissa *async*-funktioissaan ja niitä voidaan kutsua asynkronisesti. Vertailun vuoksi vielä listauksessa 14 sama sovellus toteutettuna pelkillä promiseilla.

```
orderItemShop1()
  .then(res => sellItem(res))
  .then(res => console.log('I got ' + res + ' as payment.');

orderItemShop2()
  .then(res => useItem(res));
```

Listaus 14: Demo 2 promiseilla

Listauksen 14 toteutus on selvästi yksinkertaisempi kuin toteutus *async/await*:in avulla tehdyllä, mutta tässä tulee ottaa huomioon, että kyseessä on vain yksi tietty esimerkki. Näiden esimerkkien ansiosta kuitenkin huomataan, että *async/await*:in kanssa tulee olla tarkka siinä, miten ja missä sitä käyttää, koska lopputulos voi olla hyvin erilainen pienilläkin muutoksilla.

6. YHTEENVETO

Selaimet tarjoavat vain yhden suoritussäikeen niille, jotka tarvitsevat pääsyä käyttöliittymään. Tämän takia web-ohjelmoinnissa tulee käyttää hyödyksi kaikkia selainten tarjoamia asynkronisia ominaisuuksia, jotta käyttäjäkokemus ei kärsi turhasta odottamisesta ja saadaan aikaan responsiivisia ja dynaamisia web-sovelluksia. Työssä käytiin ensin läpi web-kehityksen historian vaiheita, jotka johtivat asynkroniseen web-ohjelmointiin ja tämän jälkeen käytiin läpi asynkronisuuden toteutustapoja JavaScriptissä. Web-kehityksen vaiheet noudattavat keskenään hyvin samantyyppistä kaavaa; ensin kehittäjille mahdollistetaan jokin ominaisuus ja tämän jälkeen tehdään sen toteuttamisesta helpompaa.

Callback-metodi on kriittinen osa asynkronisuuden toteutuksessa JavaScriptissä, mutta sen luettavuuden vaikeus ja monimutkaisuus on saatu käytännölliselle ja helppolukuiselle tasolle promiseiden avulla. Promiset tekevät asynkronisuuden toteutuksesta selkeää ja helposti muokattavaa *.then()*- ja *.catch()*-metodiensa lohkojaon ja ketjutusmahdollisuuden ansiosta.

Async/await mahdollistaa asynkronisen ohjelmoinnin synkronisella tavalla estämättä pääohjelman suoritusta. *Async/await* perustuu kuitenkin täysin promise-pohjaiseen koodiin ja täten ymmärrys promiseiden toiminnasta on tarpeellista välttääkseen virheet käyttäessä *async/await*-metodia. Tämä metodi on mukavampi vaihtoehto niille, jotka haluavat kirjoittaa asynkronista koodia synkronisella rakenteella, mutta voi vaikeuttaa koodin luettavuutta ja ymmärrettävyyttä monimutkaisemmissa tapauksissa. Koodin hidastumisen riski virheellisestä *async/await*-rakenteesta on suurempi verrattuna promiseiden ketjutukseen.

Asynkronisuuden toteutusta ja uusia keinoja, jotka tarjoavat vaihtoehtoisia toteutuksia promiseille kuten Fluture [1], kehitetään edelleen. Fluture ei ole mukana tämän työn vertailuissa johtuen työn teko vaiheessa Fluturen nuoresta iästä.

LÄHTEET

[1] Aldwin Vlasblom, “Fluture - fantasy land compliant alternative to promises,” 2018, Viitattu: 25.6.2018. Saatavissa: https://github.com/fluture-js/Fluture

[2] Alexander Zlatkov, “How javascript works: Event loop and the rise of async programming + 5 ways to better coding with async/await,” 2017, Viitattu: 11.7.2018. Saatavissa: https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-cod

[3] Apigee Docs, “Asynchronous vs. synchronous calls,” 2018, Viitattu: 25.5.2018. Saatavissa: https://docs.apigee.com/api-baas/get-started/asynchronous-vs-synchronous-calls

[4] M. S. Biçer and B. Diri, “Defect prediction for cascading style sheets,” pp. 1078–1084, 2016, iD: 272229 http://www.sciencedirect.com.libproxy.tut.fi/science/article/pii/S1568494616302484

[5] B. Carter, “HTML Architecture, a Novel Development System (HANDS): An Approach for Web Development.” IEEE, 2014, pp. 90–95.

[6] K. Gallaba, A. Mesbah, and I. Beschastnikh, “Don’t Call Us, We’ll Call You: Characterizing Callbacks in JavaScript,” in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–10, iD: 1.

[7] Google Developers, “Async functions - making promises friendly,” 2018, Viitattu: 10.5.2018. Saatavissa: https://developers.google.com/web/fundamentals/primers/async-functions

[8] Håkon Wium Lie, “Cascading HTML style sheets - a proposal,” 1994, Viitattu: 31.5.2018. Saatavissa: https://www.w3.org/People/howcome/p/cascade.html

[9] Jesse James Garrett, “Ajax: A New Approach to Web Applications,” Viitattu: 1.6.2018. Saatavissa: http://experiencezen.com/wp-content/uploads/2007/04/adaptive-path-ajax-a-new-approach-to-web-applications1.pdf

[10] J. Keith and J. Sambells, *DOM Scripting: Web Design with JavaScript and the Document Object Model*, 2nd ed. Berkeley, CA: Apress, 2010.

[11] B. Lin, Y. Chen, X. Chen, and Y. Yu, “Comparison between json and xml in applications based on ajax,” in *Computer science & service system (csss), 2012 international conference on*. IEEE, 2012, pp. 1174–1177.

[12] MDN Web Docs, “Fetching data from the server,” 2018, Viitattu: 25.5.2018. Saatavissa: https://developer.mozilla.org/en-us/docs/learn/javascript/client-side_web_apis/fetching_data

[13] MDN Web Docs, “forEach,” 2018, Viitattu: 20.5.2018. Saatavissa: https://developer.mozilla.org/en-us/docs/web/javascript/reference/global_objects/array/foreach

[14] MDN Web Docs, “Promise,” 2018, Viitattu: 25.5.2018. Saatavissa: https://developer.mozilla.org/en-us/docs/web/javascript/reference/global_objects/promise

[15] MDN Web Docs, “Promise.all(),” 2018, Viitattu: 16.6.2018. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

[16] MDN Web Docs, “Promise.race(),” 2018, Viitattu: 16.6.2018. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

[17] T. Mikkonen and A. Taivalsaari, “Web applications - spaghetti code for the 21st century.” IEEE, 2008, pp. 319–328.

[18] M. Ogden, “Callback Hell,” 2015, Viitattu: 29.5.2018. Saatavissa: http://callbackhell.com

[19] R. Sarieddine, *JavaScript Promises Essentials*, 1st ed. Packt Publishing, 2014.

[20] Sunava Dutta, “Native XMLHTTPRequest object,” Viitattu: 1.6.2018. Saatavissa: https://blogs.msdn.microsoft.com/ie/2006/01/23/native-xmlhttprequest-object/

[21] W3, “Cascading Style Sheets, level 1,” Viitattu: 31.5.2018. Saatavissa: https://www.w3.org/TR/1999/REC-CSS1-19990111

[22] W3, “Document Object Model (DOM) Level 1 Specification,” Viitattu: 31.5.2018. Saatavissa: https://www.w3.org/TR/REC-DOM-Level-1/

[23] W3, “Writing Promise-Using Specifications,” 2017, Viitattu: 30.5.2018. Saatavissa: https://www.w3.org/2001/tag/doc/promises-guide

[24] W3, “World Wide Web Consortium,” 2018, Viitattu: 31.5.2018. Saatavissa: https://www.w3.org/Consortium/