Tampereen yliopisto

Salle Helevä

FUNKTIONAALINEN OHJELMOINTI JAVASCRIPTILLÄ

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Toukokuu 2021

TIIVISTELMÄ

Salle Helevä: Funktionaalinen ohjelmointi JavaScriptillä
Kandidaatintyö
Tampereen yliopisto
Tietotekniikan koulutusohjelma
Toukokuu 2021

JavaScript on ohjelmistoteollisuudessa merkittävä ohjelmointikieli. Sen pääasiallinen käyttökohde on web-pohjaisessa ohjelmistokehityksessä. JavaScript on kehittynyt selaimella ajettavasta skriptikielestä ohjelmointikieleksi, jolla toteutetaan mm. web-sivujen palvelinpuolen ohjelmistoja. Modernin JavaScriptin ominaisuudet tekevät siitä moniparadigmakielen. Tässä työssä tehdään katsaus funktionaalisen ohjelmointiparadigman mukaiseen ohjelmointiin JavaScriptillä. Katsauksessa selvitetään, miten funktionaalisella JavaScriptillä voidaan vastata sen käyttötarkoitusten tyypillisiin haasteisiin.

Moderni JavaScript noudattaa ECMAScript 2015 (ES6) -standardia, joka on ohjelmointikielenä ominaisuuksiltaan rikkaampi ja varttuneempi kuin edeltäjänsä. Työssä tarkastellaan, miten funktionaalisen ohjelmoinnin määritelmää voidaan noudattaa modernilla JavaScriptillä ohjelmoitaessa ja miten funktionaaliselle ohjelmoinnille tyypillisiä suunnittelumalleja on mahdollista toteuttaa. Funktionaalisen ohjelmoinnin tuomia etuja mahdollisissa käyttötapauksissa havainnollistetaan työssä käyttämällä yksinkertaisia esimerkkejä.

Katsauksen tuloksena huomataan, että JavaScript soveltuu hyvin funktionaaliseen ohjelmointiin. Funktionaalisen ohjelmoinnin hyödyt, kuten deklaratiivisempi syntaksi, lisääntynyt uudelleenkäytettävyys, modulaarisuus ja testattavuus, auttavat vastaamaan käytännön haasteisiin JavaScriptillä ohjelmoitaessa.

Avainsanat: funktionaalinen ohjelmointi, JavaScript, ohjelmointiparadigmat, ohjelmistokehitys

SISÄLLYSLUETTELO

1. JOHDANTO

Nykypäivän suosituimmat ohjelmointikielet mahdollistavat useiden eri ohjelmointiparadigmojen noudattamisen. Selainpuolen web-sovellusten pääasiallinen kieli JavaScript ei ole poikkeus. JavaScript on moniparadigmakieli, ja sillä on mahdollista ohjelmoida sekä deklaratiivisen että imperatiivisen paradigman mukaisesti. Deklaratiivinen ja imperatiivinen paradigma ovat yläkäsitteitä, joiden alle suosituimpina kuuluvat deklaratiivista paradigmaa edustava funktionaalinen ohjelmointi ja imperatiivista paradigmaa edustava olio-ohjelmointi. Funktionaalinen ohjelmointi sai alkunsa 1930-luvulla kehitetystä lambda-kalkyylistä, ja ensimmäinen funktionaalinen ohjelmointikieli Lisp julkaistiin vuonna 1958 [1, s. 32]. Viime vuosina funktionaalinen paradigma on saanut uutta suosiota ohjelmistokehityksen piirissä. Uudet funktionaaliset ohjelmointikielet kuten Haskell (revisio julkaistu 2010) ja Clojure (ensimmäinen julkaisu 2007), ovat tänä päivänä suurten teknologiayritysten kuten Facebookin käytössä, mikä osoittaa suosion kasvua ohjelmistoteollisuudessa. [2, s. 349] Myös useat modernin JavaScriptin (ECMAScript 2015 tai ES6) sisäänrakennetut toiminnallisuudet edustavat funktionaalisia suunnittelumalleja. Tässä työssä esiintyvät esimerkit ovat ES6-yhteensopivia ja puhuttaessa JavaScriptistä viitataan ES6-standardin mukaiseen JavaScriptiin.

Työn tarkoituksena on selvittää, miten funktionaalista ohjelmointiparadigmaa voidaan noudattaa JavaScriptillä ohjelmoitaessa. Lisäksi tehdään katsaus funktionaalisen JavaScriptin soveltuvuudesta web-kehitykseen, JavaScriptin pääasialliseen käyttökohteeseen.

Seuraavassa luvussa 2 esitellään funktionaalisen paradigman määritelmä. Sitä seuraavassa luvussa 3 käsitellään funktionaalisen ohjelmoinnin pääasiallisia tekniikoita ja suunnittelumalleja ja niiden toteuttamista JavaScriptillä. Luvussa 4 tutkitaan, miten funktionaalinen ohjelmointi vastaa käytännön vaatimuksiin web-kehityksessä. Viimeisessä luvussa on yhteenveto.

2. MÄÄRITELMÄ JA PERIAATTEET

Funktionaalinen ohjelmointiparadigma on yksi deklaratiivisista paradigmoista. Deklaratiivisissa ohjelmointiparadigmoissa suositaan abstraktioita, jolloin ohjelmoijan tehtäväksi jää halutun tuloksen määritteleminen ilman, että hän joutuu keskittymään siihen, miten operaatio on toteutettu matalammalla tasolla. Funktionaalisessa ohjelmoinnissa ohjelman toteutus rakentuu funktioiden ympärille. Deklaratiivisuus siis ilmenee uudelleenkäytettävinä funktiototeutuksina, joissa jokainen funktio toteuttaa jonkin toiminnallisuuden. Ohjelma itsessään on funktio, joka määritellään muiden funktioiden perusteella. Funktiot voivat toimia syötteenä toisille funktoille ja palauttaa funktoita paluuarvonaan. [3, ss. 23–24] Seuraava määritelmä kuvaa funktionaalisen ohjelmoinnin periaatteet ideaalitapauksessa, jossa paradigmaa noudatetaan täydellisesti, kuten puhtaasti funktionaalisella kielellä ohjelmoitaessa. JavaScript on moniparadigmakieli, joten määritelmästä on mahdollisuus poiketa tai noudattaa sitä valikoivasti sovellusalueen vaatimusten mukaan.

Funktionaalisessa ohjelmoinnissa ohjelmistokoodi muodostetaan puhtaista funktioista. Puhdas funktio vastaa funktion määritelmää matematiikassa. Funktio ottaa syötteen, toteuttaa sille jonkin operaation ja palauttaa paluuarvon. Funktion syöte kuvautuu joksikin arvoksi funktiolle mahdollisten paluuarvojen joukossa, ja samalla syötteellä saadaan aina sama paluuarvo. [4, s. 6] Useissa ohjelmointikielissä, kuten myös JavaScriptissä, on mahdollista toteuttaa funktioita, jotka eivät ole puhtaita. Tällaiset funktiot voivat siis tuottaa identtisellä syötteellä useita eri paluuarvoja riippuen jostakin funktion ulkopuolisesta tekijästä, jota ei anneta funktiolle syötteeksi. Esimerkiksi olio-ohjelmoinnissa hyödynnetään ei-puhtaita funktioita. Olio-ohjelmoinnissa oliolla on tyypillisesti jäsenmuuttujista muodostuva sisäinen tila, johon on mahdollista tehdä muutoksia luokan metodien kautta [5, ss. 216–220]. Tässä tapauksessa metodit eivät välttämättä ole puhtaita funktioita. Seuraava koodi on esimerkki yksinkertaisesta luokkatoteutuksesta Javascript-kielellä, jossa metodi `increment` ei ole puhdas funktio, sillä sen paluuarvo ei pysy samana, vaikka syöte ei muutu. Tässä tapauksessa parametreja ei ole, eli syöte on tyhjä.

```
class Counter {
  constructor() {
    this.count = 0;
  }
  increment() {
    this.count = this.count + 1;
    return this.count;
  }
}
```

2.1. *JavaScript-luokkatoteutus*

Funktionaalisessa ohjelmoinnissa tietoalkiot ovat muuttumattomia. Jos siis halutaan muuttaa ohjelman sisäistä tilaa, ainoa tapa on luoda uusi tietoalkio, johon kopioidaan aiemman tietoalkion arvo, ja tähän tehdään halutut muutokset alustamishetkellä. Syy tälle vaatimukselle on, että funktioiden puhtaana pysyminen edellyttää, ettei funktion syötteeksi annettuun parametriin tai funktion skoopin ulkopuolelle tallennettuun tietoon tehdä muutoksia. Kuitenkin funktiot voivat lukea tietoa oman skooppinsa ulkopuolelta globaalista tai korkeamman skoopin muuttujasta. Jos nämä funktion ulkopuoliset, mutta mahdollisesti sen paluuarvoon vaikuttavat, tekijät pysyvät aina muuttumattomina, funktiot voivat lukea niitä, mutta paluuarvo on silti sama samalla syötteellä. [6, s. 297][3, ss. 23–24]

Koska funktiot ovat puhtaita eli funktion paluuarvo on määritelty vain syötteen perusteella, sivuvaikutuksilta vältytään täysin. Sivuvaikutus tarkoittaa, että funktio tekee jotain muuta paluuarvon laskemisen lisäksi, esimerkiksi muuttaa globaalin muuttujan arvoa. Edellisessä esimerkissä funktion sivuvaikutus on olion jäsenmuuttujan `count` arvon muuttaminen. Tässä yksinkertaisessa esimerkkitapauksessa sivuvaikutus on helppo määritellä ja pitää silmällä, mutta ohjelman monimutkaisuuden lisääntyessä sivuvaikutusten seuraaminen muuttuu haastavaksi. [3, s. 24] On siis selvää, että funktionaalinen paradigma tarjoaa etuja, joista mainittakoon testattavuus, uudelleenkäytettävyys ja funktionaalinen riippumattomuus. [1, s. 32] Funktionaalisen ohjelmointiparadigman noudattaminen on kuitenkin eräänlainen kompromissi, kuten mikä tahansa muukin paradigmavalinta. Esimerkiksi tietoalkioiden muuttumattomuusvaatimus asettaa erityisen tiukat rajoitteet implementaatiolle, ja joissakin tapauksissa edellä mainitut hyödyt eivät välttämättä korvaa satunnaisesta säännöistä poikkeamisesta aiheutunutta ajan säästöä. JavaScriptin kaltaisen moniparadigmakielen tapauksessa paradigman periaatteiden hyödyntäminen valikoivasti onkin mahdollista.

Funktionaalisen paradigman määritelmä voidaan siis kiteyttää kahteen sääntöön: funktioiden on oltava puhtaita, ja tietoalkioiden on oltava muuttumattomia. Nyt ohjelmoijan tehtäväksi jää halutun ohjelman implementointi näitä sääntöjä noudattaen. Näiden sääntöjen seurauksena funktionaaliseen paradigmaan on vakiintunut useita suunnittelumalleja ja käytäntöjä. Suunnittelumalleissa funktiot ovat pääasiallinen tietorakenne, joiden ympärille toiminnallisuus rakennetaan. Deklaratiivisesti, funktioiden toteuttamaa toiminnallisuutta yhdistelemällä, saadaan kehitettyä lopullinen ohjelma.

3. FUNKTIONAALISEN OHJELMOINNIN TEKNIIKAT JAVASCRIPTISSA

Javascript ei ole puhtaasti funktionaalinen ohjelmointikieli. Javascript mahdollistaa muunmuassa luokkien toteuttamisen ja soveltuu yhtä lailla olio-ohjelmointiin. [7] Javascript ei myöskään aseta minkäänlaisia rajoitteita alkoiden muuttumattomuudelle tai funktioiden puhtaudelle. Kuitenkin ohjelmoijan on mahdollista asettaa nämä rajoitteet itselleen ja noudattaa edellisessä luvussa esiteltyjä funktionaalisen ohjelmoinnin periaatteita haluamassaan määrin ja tarpeensa mukaan. [6, ss. 14–16]

Funktionaaliseen ohjelmointiparadigmaan kuuluu useita tekniikoita ja suunnittelumalleja (eng. design pattern), joiden esiintymistä JavaScript-kielessä kuvataan tässä luvussa. Näiden käyttäminen koodissa helpottaa funktionaalisen ohjelmoinnin periaatteiden noudattamista. Lisäksi suunnittelumallien noudattaminen tuo standardisoidun tavan toteuttaa implementaatioita tilanteessa, jossa useat vaihtoehdot ovat mahdollisia.

3.1 Korkeamman asteen funktiot

Korkeamman asteen funktio tarkoittaa funktiota, joka ottaa parametrinaan vähintään yhden toisen funktion tai palauttaa funktion paluuarvonaan. Korkeamman asteen funktiot ovat edellytys myös muille tässä luvussa esiteltäville suunnittelumalleille. JavaScript mahdollistaa korkeamman asteen funktioiden toteuttamisen ongelmitta: JavaScriptin funktioita voidaan pääasiassa kohdella kuten mitä tahansa muutakin tietotyyppiä, antaa parametreina funktiolle ja palauttaa paluuarvoina, kuten seuraavassa esimerkissä [6, ss. 16–17].

```
function myMap(arr, fn) {
  const newArray = [];
  for(let i = 0; i < arr.length; i++) {
    newArray.push(fn(arr[i]));
  }
  return newArray;
}
```

3.1. *Korkeamman asteen funktio*

Tämä yksinkertainen map-implementaatio korkeamman asteen funktiona siis ottaa parametrina `Array`-tietorakenteen ja kutsuu mitä tahansa funktiota `fn` kerran jokaista alkiota

kohden niin, että alkio on funktion parametrina. Kutsuttujen funktioiden paluuarvot tallennetaan Array-tietorakenteeseen, joka lopuksi palautetaan.

3.2 Funktorit

Olisi kätevää, jos aiemman esimerkin `myMap`-funktion kaltainen toiminnallisuus olisi automaattisesti saatavilla Array-tietorakenteen yhteydessä. Javascript kielessä Arraylle onkin olemassa valmiiksi funktio `map`, jota voi kutsua kuin luokan metodia. Tärkeää on huomata, että `map` palauttaa rakenteeltaan samanlaisen, tässä tapauksessa myös yhtä pitkän, Arrayn. Tällainen toiminnallisuus siis mahdollistaa jonkin Array-tietorakenteen kuvaamisen johonkin toiseen Array-tietorakenteeseen, joista jälkimmäisen määrittää funktio, joka annetaan korkeamman asteen `map`-funktiolle parametrina. Kyseistä suunnittelumallia kutsutaan funktoriksi. Yleisemmin siis funktori tarkoittaa kuvausta jostakin joukosta C toiseen joukkoon D seuraavaavan määritelmän mukaisesti [6, s. 374-375]:

- 1. Jokainen joukon C alkio x kuvataan alkioksi $fn(x)$ joukkoon D.
- 2. Mikäli funktio $fn(x)$ on identiteettifunktio, silloin $x = fn(x)$.
- 3. Kaksi peräkkäistä kuvausta tuottavat saman tuloksen kuin niiden kompositio.

Arrayn metodi `map` toteuttaa näistä jokaisen säännön. Kohta 1 lienee itsestäänselvä. Kohta 2 täyttyy, sillä jos `map`-funktion parametriksi annetaan identiteettifunktio eli pelkkä alkio seuraavasti,

```
myArray.map(x => x)
```

tuloksena on kopio alkuperäisestä tietorakenteesta.

Kohta 3 toteutuu sillä

```
myArray.map(fn1).map(fn2)
```

tuottaa saman paluuarvon kuin seuraava:

```
myArray.map((x) => {return fn2(fn1(x))})
```

Array tietorakenteella on `map`-funktion lisäksi myös muita hyödyllisiä metodeita, kuten `filter`, `reduce`, `find` yms. Näiden käyttäminen Array-tietorakenteen kanssa ei kuitenkaan tarkalleen täytä funktorin määritelmää, sillä edellä mainitut ehdot eivät aina toteudu. Kuitenkin periaate on samanlainen: tietorakenteen alkioita kuvataan toiseen tietorakenteeseen korkeamman asteen funktion ja sille parametrina annetun funktion määräämällä tavalla. JavaScript mahdollistaa myös metodien ketjuttamisen, kuten aiemmin kahden peräkkäisen `map`-funktiokutsun tapauksessa, joka tukee intuitiivisen syntaksin toteuttamista funktionaalista paradigmaa noudattavalle ohjelmoijalle ja säilyttää tilan ketjun sisäpuolella, jolloin ulkopuoliseen tilaan tehdyiltä sivuvaikutuksilta vältytään [8, s. 39-41]. Funktorit

mahdollistavat siis datan manipuoloimisen ohjelman sisällä ilman alkuperäisen tietoalkion arvon muuttamista tai muita sivuvaikutuksia. [6, s. 376]

3.3 Monadit

Monadi on funktionaaliselle ohjelmoinnille tyypillinen suuunnittelumalli, jolla on mahdollista luoda toiminnallisuutta, joka muussa tapauksessa johtaisi sivuvaikutuksiin. Monadi kapseloi sisälleen joitakin alkioita ja tarjoaa rajapinnan operaatioiden suorittamiseen, kuitenkin säilyttäen itse arvot ja mahdolliset sivuvaikutukset sisäisessä kontekstissaan. Puhtaasti funktionaalisissa kielissä esimerkiksi satunnaislukujen generointi tai I/O-operaatioiden käsittely ovat monadille tyypillisiä käyttökohteita, sillä niihin implisiittisesti liittyviltä sivuvaikutuksilta voidaan monadien avulla välttyä. [2, ss. 358–359]

Monadi on myös funktori ja noudattaa aiemmin esitetyn funktorin määritelmää. [6, ss. 384–388]. Monadille on lisäksi määritelty yksi pakollinen operaatio `bind`, joka nostaa parametrina annetun alkion monadin omaan kontekstiin, suorittaa sille halutun operaation ja palauttaa uuden monadin, jonka sisälle on kapseloitu suoritetun operaation tulos [2, s. 358]. Esimerkiksi yksinkertainen identiteettimonadi havainnollistaa `bind`-operaation toteutuksen:

```
function Identity(value) {
  this.value = value;
}

Identity.prototype.bind = function(transform) {
  return transform(this.value);
};

var result = new Identity(5).bind(value =>
    new Identity(6).bind(value2 =>
      new Identity(value + value2)
    )
  );
```

3.2. *Identiteettimonadi. Lähde ks. [9].*

Esimerkissä `Identity`-monadin `bind`-funktio tarjoaa siis rajapinnan operaatioille, joissa käytetään monadin sisälle kapseloitua arvoa `this.value`. Funktion `bind` paluuarvo on uusi monadi, jonka kapseloimaan alkioon `this.value` on alustettu operaation tulos. Kutsumalla `console.log(result)` tulostuu `Identity`-funktio, jonka jäsenmuuttujan `this.value` arvo on 11 eli yhteenlaskun tulos.

Toinen yleinen monadinen suunnittelumalli on niinsanottu Maybe-monadi. Maybe-monadi on abstrakti tietotyyppi, joka tuo funktionaalisen paradigman mukaisen ratkaisun paluuarvoltaan epävarmojen operaatioiden suorittamiseen [10, ss. 27–29]. Maybe-monadi voidaan implementoida esimerkiksi JavaScriptin funktioiden avulla seuraavasti:

```
function Maybe(value) {
  if (value instanceof Maybe) return value;

  if (this instanceof Maybe) {
    if (value === null || value === undefined || value instanceof
       Nothing )
      return new Nothing()
    else
      this.unit = value
  } else return new Maybe(value);
}

Maybe.prototype.evaluate = function () {
  const evaluated = typeof this.unit === "function"
    ? this.unit()
    : this.unit;
  return evaluated
}

Maybe.prototype.bind = function (transform) {
  return Maybe(transform(this.evaluate()))
}

Nothing = function () {};
Nothing.prototype.bind = () => new Nothing();
Nothing.prototype.evaluate = () => new Nothing();
```

3.3. *Maybe-monadin implementaatio*

Maybe-monadi mahdollistaa poikkeustilanteiden kannalta turvallisen funktionaalisen koodin toteuttamisen. Tämä implementaatio ottaa huomioon JavaScriptille tyypillisen ongelman, jossa jonkin funktiokutsun mennessä pieleen paluuarvona on määrittämätöntä tai tyhjää arvoa edustava `undefined` tai `null`. Ohjelman suoritus jatkuu siitä huolimatta, jolloin, ilman erillistä virheenkäsittelyä jokaisessa tapauksessa, ohjelman toimintaa ei ole määritely. Ketjuttamalla funktioita `bind`-metodin avulla, mikäli jokin funktioista palauttaa arvon `null` tai `undefined`, koko ketjun suoritus lakkaa, ja lopullinen paluuarvo on tyyppiä `Nothing`. Näin ollen koodiin ei tarvitse erikseen lisätä tarkistuksia virhetapausten paluuarvoille if-else -rakenteita käyttäen; kyseinen logiikka on abstraktoitu monadin sisäiseen toteutukseen. [10, ss. 27–29] Maybe-monadilla voi korvata esimerkiksi seuraavan tyyliltään imperatiivisen toteutuksen:

```
let token;
let data = getDataFromServer();
if (!data) {
    throw Error;
} else {
    if (!payload.token) {
        throw Error;
    } else {
        token = payload.token;
    }
}
```

3.4. *Imperatiivinen poikkeustilanteiden hallinta ilman Maybe-monadia*

Sama ohjelma voidaan toteuttaa käyttämällä Maybe-monadia kuten seuraavassa esimerkissä:

```
const tokenM = Maybe(getDataFromServer())
    .bind((data) => data.payload)
    .bind((payload) => payload.token);

if (tokenM instanceof Nothing) throw Error;
```

3.5. *Poikkeustilanteiden hallinta Maybe-monadia hyödyntäen*

Monadien avulla voidaan korvata imperatiivinen lähestymistapa monimutkaisen toiminnallisuuden tapauksessa deklaratiivisemmalla tavalla.

3.4 Rekursio

Rekursiivinen funktio tarkoittaa funktiota, joka kutsuu itseään. Puhtaasti funktionaaliset kielet tarvitsevat rekursiota muuttumattomuusperiaatteen asettamien rajoitusten takia. Esimerkiksi JavaScriptin for-silmukka rikkoo muuttumattomuusperiaatetta.

```
  for(let i = 0; i < 10; i++) {
    /* ... */
  }
```

3.6. *for-silmukka*

Muuttujan `i` arvoa muutetaan jokaisella kierroksella silmukassa. Vastaava toiminnallisuus voidaan toteuttaa rekursiivisella funktiolla.

```
const forRecursive = (current, end, func) => {
    if (current === end) return;
    func(current);
    forRecursive(current + 1, end, func);
};

forRecursive(0, 10, (i) => {
    /* ... */
});
```

3.7. *Rekursiivinen implementaatio for-silmukalle*

Puhtaasti funktionaalisessa kielessä iterointi on toteutettava rekursiota hyödyntäen. Rekursiivinen iterointi JavaScriptillä voi kuitenkin olla esimerkki tilanteesta, jossa ohjelmoija hyötyy vapaudestaan rikkoa muuttumattomuusperiaatetta. Iteraattorimuuttujaan `i` tehdyt muutokset on rajoitettu silmukan skoopin sisälle. Kun pidetään huolta, ettei muuttujan `i` arvon muuttaminen aiheuta muita sivuvaikutuksia, on rekrusiivinen versio identtinen toteutus tavalliseen `for`-silmukkaan. Lisäksi JavaScriptista puuttuu useimpien selainten tapauksessa häntäkutsujen optimointi (eng. Tail Call Optimization), minkä seurauksena suuri määrä sisäkkäisiä funktiokutsuja voi johtaa kutsupinon ylivuotoon. Rekursiosta on kuitenkin hyötyä muissa käyttökohteissa, kuten esimerkiksi puutietorakenteen käsittelyyn liittyvissä algoritmeissa. [6, s. 283]

4. FUNKTIONAALINEN JAVASCRIPT WEB-KEHITYKSESSÄ

Alun perin web-sivut olivat pääasiassa staattisia HTML-dokumentteja, joiden selainpuolen logiikan monimutkaisuus rajoittui pitkälti HTML-kielen ominaisuuksiin. Selainpuolen tehtäväksi jäi vain palvelimelta pyydetyn staattisen HTML-tiedoston esittäminen käyttäjälle. Selaimilla ajettavan JavaScriptin, selainohjelmiston ja laitteiden kehityksen myötä monimutkaisenkin ohjelmiston toteuttaminen web-sovelluksena on kuitenkin nykyään mahdollista. Modernit web-sovellukset jakavat työtaakkaa palvelimen ja selaimen välillä tasaisemmin hyödyntäen asiakas-palvelin -arkkitehtuuria (eng Client-Server), jossa selain kommunikoi palvelimen kanssa, lähettää sille kutsuja usein varsin monimutkaisessa käyttöliittymälogiikassa prosessoidun syötteen mukaisesti ja selainpuolen sovelluksen tila muuttuu jotenkin palvelimelta saadun datan perusteella. [11] Tässä luvussa tutkitaan, miten funktionaalinen ohjelmointiparadigma vastaa modernin asiakas-palvelin -mallin web-sovellusten haasteisiin.

4.1 Asynkronisuus

Selain-palvelin -mallia noudattavat modernit web-sovellukset tekevät pyyntöjä palvelimelle ja operoivat vastauksesta saadulla datalla. Palvelimella kuitenkin kestää jonkin aikaa vastata asiakkaan selaimesta lähetettyyn pyyntöön. Tätä varten selaimessa on käytettävä asykronisuutta, jota varten JavaScript-kieleen on toteutettu joitakin erityispiirteisiä ominaisuuksia.

Asynkronisuus on tyypillinen käyttötapaus korkeamman asteen funktioille, metodien ketjuttamiselle ja monadisille rakenteille. Asynkronisuuden aiheuttamiin ongelmiin on mahdollista hyödyntää monadista suunnittelumallia. Asynkronisuuden lisääntyessä koodissa joudutaan ottamaan huomioon useampia tapahtumahaaroja ja koodin luettavuuden tärkeys korostuu. Abstraktoimalla asynkronisten kutsujen odottamista ja palvelimen lähettämän vastauksen eri vaihtoehtojen haarautumista jonkin rajapinnan taakse, saadaan modulaarisempaa ja luettavampaa koodia. ES6 standardin mukaisen JavaScriptin sisältämä `Promise`-tietorakenne on monadi [6, s. 392]. Se käärii asynkronisen kutsun paluuarvon sisäiseen kontekstiinsa samalla periaatteella kuin luvun 3.3 esimerkkien `Maybe`- ja `Identity`-monadit.

Palvelimelle on mahdollista tehdä pyyntö `Promise`-tietorakennetta hyödyntäen esimerkiksi seuraavalla tavalla.

```
const response = fetch(SERVER_URL)
    .then((res) => {
        return { data: res.payload };
    })
    .catch((err) => {
        return { error: err.message };
    });
```

4.1. *Asynkroninen pyyntö palvelimelle*

Funktio `fetch` palauttaa `Promise`:n. Seuraavien rivien funktioiden kutsuketju määrittelee lopullisen vakioon `response` alustettavan arvon. Mahdolliset sivuvaikutukset hallitaan korkeamman asteen funktoiden `then` ja `catch` parametrifunktioissa. Vakion `response` arvo ei käytännössä pysy muuttmattomana, vaan `fetch`-funktion kutsumishetkellä sen tyyppi on implisiittisesti arvoltaan tuntematon `Promise`, kunnes palvelin vastaa onnistuneesti tai törmätään virhetilanteeseen, jolloin lopullinen arvo määräytyy funktion `then` tai `catch` parametrifunktioiden toteutuksen perusteella [12].

Monadista suunnittelumallia voidaan hyödyntää asykronisten funktioiden kompositiossa kun halutaan yhdistää useita `Promise`-tyyppiä palauttavia funktiota yhdeksi korkeamman tason abstraktion toiminnon toteuttavaksi funktioksi. Esimerkiksi tapauksessa, jossa palvelinpuolelta joudutaan tekemään useita asynkronisia kutsuja eri rajapintoihin jonkin asiakkaan tekemän pyynnön toteuttamiseksi, voi asynkronisten funktioiden kompositiosta olla hyötyä.

```
const getUserId = (token) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            token === "exampleToken"
              ? resolve(123)
              : reject({ error: true });
        });
    });
};

const getUserData = (id) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            id === 123
              ? resolve({ data: 1 })
              : reject({ error: true });
        });
    });
};
const userDataFromToken = (token) => {
    getUserId(token)
        .then((result) => {
            getUserData(result).then(/* ... */).catch(/* ... */);
        })
        .catch(/* ... */);
};
```

4.2. *Asynkronisten funktioiden yhdistäminen ilman kompositiomonadia*

Funktiossa `userDataFromToken` kaksi aiempaa asynkronista funktiokutsua yhdistetään käyttämällä `Promise`:n `then`-metodia. Tuloksena on kutsuketju, jonka luettavuus vähenee ja toisteisuus kasvaa, mitä enemmän asykronisia kutsuja yhdistetään. Tässä tapauksessa niitä on vain kaksi, mutta oikeissa käyttötapauksissa mahdollisesti paljon enemmän. Vaihtoehtoinen ratkaisu voidaan toteuttaa käyttämällä asynkronisten funktioiden kompositioon tarkoitettua monadia:

```
composeM = (chainMethod) => (...ms) =>
    ms.reduce((f, g) => (x) => g(x)[chainMethod](f));
composePromises = composeM("then");

const userDataFromToken = (token) =>
    composePromises(getUserData, getUserId)(token)
        .then(/* ... */)
        .catch(/* ...*/);
```

4.3. *Asynkronisten funktioiden kompositio. Lähde ks. [13]*

Monadi `composeM` ottaa parametrinaan metodin, jota käytetään funktioiden kompositioon. Funktio `composePromises` ottaa parametrinaan yhdistettävät funktiot. Toteutus on toiminnaltaan identtinen sitä edeltäneeseen esimerkkiin, mutta koodissa on vähemmän toisteisuutta.

4.2 Yksikkötestaaminen

Testien kirjoittaminen ohjelmistoprojektille helpottaa kehitysprosessia. Kun lähdekoodiin tehdään muutoksia, on ilman kattavia testejä vaikea tietää, toimiiko kaikki varmasti yhä halutulla tavalla, vai toivatko muutokset tullessaan bugeja. Koska tämän työn näkökulma on varsin matalalla ohjelmointiparadigmatasolla, tehdään nyt katsaus vain yksikkötesteihin, integraatio- tai toiminnallisuustesteihin puuttumatta. Testien tarve modernissa web-kehityksessä korostuu lisääntyvän monimutkaisuuden ja asynkronisuuden myötä. Ohjelmoijan on syytä huomioida, että jokainen kutsu palvelimelle voi esim. yhteysongelman tapauksessa johtaa virhetilanteeseen.

Funktionaalinen ohjelmointiparadigma tuo puhtaiden funktioiden johdosta selkeän edun yksikkötestien kirjoittamiseen. Kattavien yksikkötestien toimivuus perustuu siihen, että testin kirjoittaja määrittää suurelle joukolle mahdollisia syötteitä tietyt paluuarvot. Jos syöte vastaa odotettua paluuarvoa, testi katsotaan läpäistyksi. Muussa tapauksessa koodissa on virhe. Koska funktionaalisen ohjelmoinnin puhtaat funktiot eivät tuota sivuvaikutuksia, funktiolle testejä kirjoittaessa ei ole otettava huomioon funktion skoopin ulkopuolisia tekijöitä. Imperatiivisemmassa lähestymistavassa on siis huomioitava kaikki mahdolliset sivuvaikutukset, jotta saadaan määriteltyä lähtötilanne ja oletettu lopputulos testattavalle toiminnallisuudelle. [1, p. 34] Esimerkiksi olio-ohjelmointiin perustuvan koodikannan luokkia testatessa luokkien sisäisen tilan on vastattava haluttua lähtötilannetta. Tämän lähtötilanteen saavuttamiseksi joudutaan mahdollisesti kutsumaan muita luokan metodeja tai alustamaan toisia olioita, jotka ovat vuorovaikutuksessa testatun luokan kanssa. Lisäksi halutun lopputilanteen määrittely on imperatiivisessa tavassa haastavampaa, sillä kaikki mahdolliset sivuvaikutukset on otettava huomioon. Funktionaalisessa koodissa lopputulos kuvautuu ainoastaan funktion paluuarvoksi.

Kuten testiympäristössä, myös todellisessa ohjelmassa funktion toiminta on riippumaton sen ulkopuolisesta skoopista. Funktionaalinen ohjelmointi siis helpottaa ohjelman jakamista pieniin osiin, joista jokaisen toimintaa voidaan testata erikseen.

4.3 Modulaarisuus ja uudelleenkäytettävyys

Modulaarisuudella tarkoitetaan ohjelmistokehityksessä ohjelman komponenttien rakentamista moduuleista eli itsenäisesti toimivista ja jonkin yhden selkeästi määritellyn toiminnallisuuden kapseloivista rakennuspalasista. Samaan käsitteeseen sisältyy läheisesti uudelleenkäytettävyys. Yhtä moduulia on mahdollista käyttää useassa eri kontekstissa sen riippumattomuudesta johtuen. Modulaarisuus on oleellinen osa modernia web-kehitystä. Lukuisat komponenttikirjastot nopeuttavat kehitysprosessia tarjoamalla valmiiksi toteutettuja, laajalti käytettyjä ja testattuja modulaarisia ratkaisuja web-sovellusten kehittäjille. JavaScript-moduuleihin keskittyvät paketinhallintaohjelmat kuten NPM (Node Package Manager) tuovat kehittäjien saataville satoja tuhansia JavaScript-moduuleja. Myös yksittäisen projektin sisällä suunnittelemalla ohjelmistoa modulaarisuus ja uudelleenkäytettävyys silmällä pitäen, voidaan välttyä merkittävältä teknologisen velan määrältä. Koska modulaarisuus edellyttää ohjelmiston rakentamista itsenäisesti toimivista, toisistaan riippumattomista moduuleista, sopii funktionaalinen ohjelmointiparadigma hyvin modulaarisuuden toteuttamiseen. Jokainen funktio kapseloi yhden toiminnallisuuden. Koska funktiot ovat puhtaita, on niitä mahdollista käyttää useassa eri kontekstissa, sillä funktion paluuarvo ei riipu ympäröivän ohjelman tilasta vaan ainoastaan funktiolle syötetyistä parametreista.

Olio-ohjelmoinnissa yksittäinen olio voi muodostaa moduulin, mutta koska usein toiminnan toteuttamiseksi vaaditaan olioiden välistä vuorovaikutusta, sopivan abstraktiotason valitseminen moduulille voi olla haastavaa [3, s. 363]. Funktionaalisessa ohjelmoinnissa moduuli toteutetaan funktiona kuten muutkin toiminnallisuudet. Funktioiden puhtaus takaa sen, että funktion ulkopuoliset tekijät eivät vaikuta sen lopputulokseen ja tuloksena on täysin itsenäinen ja modulaarinen komponentti. Tämä funktionaalinen riippumattomuus mahdollistaa funktioiden käyttämisen missä tahansa kontekstissa. Nyt funktio abstraktoi sisälleen jonkin toiminnallisuuden ja ohjelmoijan tarvitsee tietää toteutuksesta vain, millaisella syötteellä se toimii, ja millaista tietoa funktio palauttaa. Siksi on tärkeää määrittää funktiolle rajapinta, joka kuvaa sille syötettävät parametrit, parametrien tyypit ja mahdolliset paluuarvot. Näin funktion toiminnan oleellisin osa on dokumentoitu sen uudelleenkäyttöä varten. Yksi standardi rajapinnan määrittelyyn ovat Hindley–Milner tyyppisignatuurit, joita mm. funktionaalinen ohjelmointikieli Haskell käyttää tyyppitarkistukseen. JavaScript-funktioiden rajapintojen määrittelyyn voidaan käyttää esimerkiksi JSDoc-standardia. Myös JavaScriptiksi kääntyvä TypeScript mahdollistaa tyyppien määrittelyn ja tyyppien staattisen tarkistuksen, mikä tekee rajapintojen dokumentoinnista ja noudattamisesta helpompaa. [6, s. 365–356] [1, ss. 34–35]

5. YHTEENVETO

JavaScriptin tekninen toteutus mahdollistaa funktionaalisen ohjelmointiparadigman kannalta oleellisten vaatimusten täyttämisen. Kieli kohtelee funktioita kuten muitakin tietorakenteita, mistä johtuen funktionaaliset suunnittelumallit, kuten korkeamman asteen funktiot, funktorit ja monadit, on helppo toteuttaa. Vaikka JavaScript mahdollistaa myös funktionaalisen paradigman sääntöjen, kuten muuttumattomuuden ja puhtaiden funktioiden, rikkomisen on ohjelmoijan silti mahdollista asettaa nämä säännöt itselleen tai poiketa niistä harkintansa mukaan. Funktionaalisen paradigman periaatteita ja suunnittelumalleja noudattamalla voidaan hyötyä funktionaalisen ohjelmoinnin tuomista eduista JavaScript-ohjelmien kehitysprosessissa.

Modernin web-kehityksen vaatimusten myötä deklaratiivisesta lähestymistavasta saadut hyödyt korostuvat ja haasteisiin on mahdollista vastata funktionaalista JavaScriptia käyttäen. Funktionaalisen ohjelmointiparadigman noudattaminen tuo etuja imperatiivisempaan lähestymistapaan verrattuna. Koodista on helppo vähentää toisteisuutta ja luettavuus paranee. Koodikannan osista on mahdollista tehdä modulaarisia, uudelleenkäytettäviä funktioita, joiden yksikkötestaaminen on helppoa. Modernien web-sovellusten monimutkaista logiikkaa on mahdollista toteuttaa funktionaalisia suunnittelumalleja ja tekniikkoja hyödyntäen.

JavaScript on siis soveltuva funktionaaliseen ohjelmointiin. Funktionaalinen ohjelmointiparadigma tarjoaa vaihtoehtoisen lähestymistavan ongelmanratkaisulle perinteisen imperatiivisen olio-ohjelmoinnin rinnalle.

LÄHTEET

[1] Benton ja Radziwill. Improving Testability and Reuse by Transitioning to Functional Programming (2016).

[2] Hu, Hughes et al. How functional programming mattered. *National Science Review* 2.3 (2015).

[3] Kaisler. *Software Paradigms*. Wiley, 2005.

[4] Field ja Harrison. *Functional Programming*. International Computer Science Series, 1988.

[5] Lafore. *Object-Oriented Programming in C++*. Sams, 2002.

[6] Kereki. *Mastering JavaScript Functional Programming 2nd edition*. Packt, 2020.

[7] *Javascript reference > Classes*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes. (accessed: 19.01.2021).

[8] Mansilla. *Reactive Programming with RxJS 5*. Pragmatic Bookshelf, 2018.

[9] *Curiosity driven > Monads in JavaScript*. URL: https://curiosity-driven.org/monads-in-javascript. (accessed: 19.01.2021).

[10] Spivey. A Functional Theory of Exceptions. *Science of Computer Programming* (1990).

[11] Saternos. *Client-Server Web Apps with JavaScript and Java*. O'Reilly, 2014.

[12] *Javascript reference > Standard Built-in Objects > Promise*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global%5C_Objects/Promise. (accessed: 30.01.2021).

[13] Elliot. *JavaScript Monads Made Simple*. URL: https://medium.com/javascript-scene/javascript-monads-made-simple-7856be57bfe8. (accessed: 7.3.2021).