
Funktionaalinen ohjelmointi front-end web-kehityksessä

TkK-tutkielma
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Ohjelmistotekniikka
2020
Konsta Purtsi

TURUN YLIOPISTO

Tulevaisuuden teknologioiden laitos

KONSTA PURTSI: Funktionaalinen ohjelmointi front-end web-kehityksessä

TkK-tutkielma, 18 s.

Ohjelmistotekniikka

Kesäkuu 2020

Tarkempia ohjeita tiivistelmäsivun laadintaan löytyy opiskelijan yleisoppaasta, josta alla lyhyt katkelma.

Bibliografisten tietojen jälkeen kirjoitetaan varsinainen tiivistelmä. Sen on oletettava, että lukijalla on yleiset tiedot aiheesta. Tiivistelmän tulee olla ymmärrettävissä ilman tarvetta perehtyä koko tutkielmaan. Se on kirjoitettava täydellisinä virkkeinä, väliotsakeluettelona. On käytettävä vakiintuneita termejä. Viittauksia ja lainauksia tiivistelmään ei saa sisällyttää, eikä myöskään tietoja tai väitteitä, jotka eivät sisälly itse tutkimukseen. Tiivistelmän on oltava mahdollisimman ytimekäs n. 120–250 sanan pituinen itsenäinen kokonaisuus, joka mahtuu ykkäsvälillä kirjoitettuna vaivatta tiivistelmäsivulle. Tiivistelmässä tulisi ilmetä mm. tutkielman aihe tutkimuksen kohde, populaatio, alue ja tarkoitus käytetyt tutkimusmenetelmät (mikäli tutkimus on luonteeltaan teoreettinen ja tiettyyn kirjalliseen materiaaliin, on mainittava tärkeimmät lähdeteokset; mikäli on luonteeltaan empiirinen, on mainittava käytetyt menetit) keskeiset tutkimustulokset tulosten perusteella tehdyt päätelmät ja toimenpidesuosituksat asiasanat

Asiasanat: tähän, lista, avainsanoista

Sisältö

1	Johdanto	1
1.1	Motivaatio	1
1.2	Tutkimuskysymys	1
1.3	Metodit	1
1.4	Rakenne	1
2	Funktionaalinen ohjelmointi	2
2.1	Määritelmä	2
2.2	Lambdakalkyyli	3
2.3	Konseptit	3
2.3.1	Tilattomuus	3
2.3.2	Korkeamman asteen funktiot	4
2.3.3	Laiska laskenta	5
2.3.4	Hahmonsovitut	6
2.4	Funktionaalisen laskentamallin seuraukset	7
3	Web-kehitys	8
3.1	World Wide Web	8
3.1.1	Staatittiset verkkosivut	8
3.1.2	Dynaamiset verkkosivut	9
3.2	Back-end	9

3.3	Front-end	10
3.4	Funktionaalinen web-ohjelmointi	10
3.5	Funktionaalinen reaktiivinen ohjelmointi	12
4	Funktionaaliset front-end ohjelmistokehykset	13
4.1	Ohjelmistokehysten esittely	13
4.2	Funktionaalinen ohjelmointi	14
4.3	Funktionaalinen tilankäsittely	14
4.4	Sivuvaikutukset	17
4.5	Funktionaalinen tapahtumankäsittely	17
5	Loppupäätelmät	18
	Lähdeluettelo	19

1 Johdanto

1.1 Motivaatio

1.2 Tutkimuskysymys

1.3 Metodit

1.4 Rakenne

2 Funktionaalinen ohjelmointi

2.1 Määritelmä

Funktionaalinen ohjelmointi on deklarativinen ohjelmointiparadigma, jossa ohjelman suoritus tapahtuu sisäkkäisten matemaattisten funktioiden evaluomisena. Deklaratiivinen ohjelmointi on ohjelmointiparadigma, jossa kuvaillaan ohjelman lopputulos tai tila. Deklaratiivisessa ohjelmoinnissa ei ole epätarkkaa globaalia tilaa, ja siten ohjelma koostuu lausekkeista, jotka voivat sisältää lokaalia epätarkkaa tilaa. Se on vastakohta perinteisemmälle imperatiiviselle ohjelmoinnille. Funktionaalinen ohjelmakoodi pyrkii käsittelemään tilaa tarkemmin ja välttämään muuttuvia arvoja. [1]

Puhtaasti funktionaalinen ohjelmointikieli ei salli lainkaan muuttuvaa tilaa tai datan muokkaamista. Tämä vaatii sen, että funktioiden paluuarvo riippuu ainoastaan funktion saamista parametreista, eikä mikään muu kuin parametrit voi vaikuttaa funktion paluuarvoon. Funktionaaliseksi ohjelmointikieleksi lasketaan kuitenkin myös moniparadigmaiset ohjelmointikielet, jotka sisältävät funktionaalisia piirteitä. Yksinkertaisten loogisten toimintojen lisäksi realistinen ohjelmointikieli vaatii epäpuhtauksia muun muassa siirrännän (eng. Input/Output) toteuttamiseen. [2] Moderneissa ohjelmointikielissä tähän on myös funktionaalisia ratkaisuja, esimerkiksi Haskell-kielen IO-monadi [3].

2.2 Lambdakalkyyli

Funktionaalinen ohjelmointi perustuu lambdakalkyyliin - formaalin laskennan malliin jonka kehitti Alonzo Church vuonna 1928. Lambdakalkyyliä pidetään ensimmäisenä funktionaalisena ohjelmointikielenä, vaikka se keksittiinkin ennen ensimmäisiä varsinaisia tietokoneita, eikä sitä pidettykään aikanaan ohjelmointikielenä. Useita moderneja ohjelmointikieliä, kuten Lispä, pidetään abstraktioina lambdakalkyylin päälle. Lambdakalkyyli voidaan kirjoittaa myös λ -kalkyyli kreikkalaisella lambda-kirjaimella. [1]

Churchin lambdakalkyyli koostuu vain kolmesta termistä: muuttujista, abstraktioista ja sovelluksista. Muuttujat ovat yksinkertaisesti merkkejä tai merkkijonoja jotka kuvaavat jotain parametriä tai arvoa. Churchin alkuperäinen lambdakalkyyli ei tuntenut muuttujien asettamista, ainoastaan arvojen syöttämisen parametrina. Abstraktio $\lambda x[M]$ määrittelee funktion parametrille x toteutuksella M . Sovellus $\{F\}(X)$ suorittaa funktion F arvolla X . [4]

2.3 Konseptit

2.3.1 Tilattomuus

Deklaratiivisissa ohjelmointikielissä ei ole implisiittistä tilaa, mikä eroaa imperatiivisista kielistä, joissa tilaa voi muokata lauseilla (eng. commands) ohjelmakoodissa. Tilattomuus tekee laskennasta mahdollista ainoastaan lausekkeilla (eng. expression). Funktionaaliset ohjelmointikielet käyttävät lambda-abstraktioita kaiken laskennan perustana. Esimerkiksi logiikkaohjelmoinnissa vastaava rakenne on relaatio. Tilattomuus sallii vain muuttumattomat vakiot, mikä estää muuttujien arvon muuttamisen ja perinteiset silmukat. Puhtaasti funktionaalisessa ohjelmoinnissa rekursio on ainoa tapa toteuttaa toisto. Esimerkiksi funktio, joka laskee luvun n kertoman voitaisiin

toteuttaa imperatiivisesti Python-kielessä esimerkiksi näin:

```
def factorial(n):  
    result = 1  
    while n >= 1:  
        result = result * n  
        n = n - 1  
    return result
```

Funktionaalisessa ohjelmointikielessä toistorakenne on toteutettava rekursion avulla, mutta tästä on usein tehty vaivatonta muun muassa yksinkertaisen syntaksin ja tehooptimoinnin avulla. [1] Funktio, joka laskee luvun n kertoman voidaan toteuttaa esimerkiksi näin funktionaalisesti Haskell-kielessä:

```
factorial :: Int -> Int  
factorial n = if n < 2  
    then 1  
    else n * factorial (n-1)
```

2.3.2 Korkeamman asteen funktiot

Funktionaaliset ohjelmointikielet helpottavat funktionaalista ohjelmointityyliä muun muassa sallimalla korkeamman asteen funktiot. Ensimmäisen luokan kansalainen on ohjelmointikielen rakenne, jonka voi syöttää parametrina funktioille, asettaa paluuarvoksi funktiolle ja tallentaa tietorakenteisiin. Korkeamman asteen funktio eroaa tavallisesta funktiosta siten, että se voi ottaa parametrinaan tai palauttaa funktion. Ohjelmointikielissä, jotka tukevat korkeamman asteen funktioita, funktiot ovat ensimmäisen luokan kansalaisia. Korkeamman asteen funktioiden avulla ohjelmakoodi ja data ovat jossain määrin vaihdettavissa, joten niiden avulla voidaan abstrahoida kompleksisia rakenteita. [1] Map-funktio on korkeamman asteen funktio, joka suorittaa parametrina annetun funktion jollekin tietorakenteelle, esimerkiksi listalle.

Map-funktiota käytetään usein modernissa front-end web-ohjelmoinnissa [5]. Listan renderöinti React-kirjastolla voidaan toteuttaa näin map-funktion avulla:

```
items.map(item => <Item {...item} />)
```

2.3.3 Laiska laskenta

Laiska laskenta (eng. lazy evaluation) on funktion laskentastrategia, jossa lausekkeen arvo lasketaan vasta kun sitä ensimmäisen kerran tarvitaan, mutta ei aikaisemmin. Laiska laskenta on päinvastainen laskentastrategia innokkaaseen laskentaan (eng. eager evaluation), jossa lausekkeen arvo lasketaan heti ensimmäisellä kerralla, kun lauseke esitellään. Yleisimmät ohjelmointikielet käyttävät innokasta laskentastrategiaa. Laiska laskenta vähentää suoritukseen kuluvaan aikaa ja siten voi parantaa ohjelman suorituskykyä eksponentiaalisesti esimerkiksi call by name -laskentastrategiaan verrattuna, jossa lausekkeen arvo voidaan joutua laskemaan useita kertoja. Laiskan laskennan toteuttavat useat puhtaasti funktionaaliset ohjelmointikielet, kuten Haskell. Myös jotkin moniparadigmaiset kielet toteuttavat laiskan laskennan, esimerkiksi Scala-kieli mahdollistaa sen *lazy val*-lausekkeen avulla. [6]

Laiska laskenta mahdollistaa päättymättömät tietorakenteet. Niin sanottujen laiskojen listojen loppupäää evaluoidaan vasta kun sitä kutsutaan. Tämä näkyy useissa funktionaalisissa ohjelmointikielissä listan toteutuksena, jossa listaa evaluoidaan alkupäästä loppua kohti. Esimerkiksi Haskellissa voidaan määrittää lista kaikista kokonaisluvusta tietotyypin rajoissa alkaen luvusta n :

```
from :: Int -> [Int]
from n = n : from (n+1)
```

Funktion rekursiivinen kutsu aiheuttaisi innokkaasti laskevissa ohjelmointikielissä päättymättömän rekursion, mutta Haskellin tapauksessa vain määrätty osa listasta evaluoidaan. [6]

Funktionaalisissa ohjelmointikielissä häntärekursio ei aiheuta pinon ylivuotoa, joten päättymätön rekursio on toimiva vaihtoehto päättymättömälle silmukalle. Esimerkiksi Haskell-kielen main-funktioita voidaan toistaa äärettömästi näin:

```
main :: IO ()
main =
  do
    putStrLn "do something"
  main
```

2.3.4 Hahmonsovitus

Hahmonsovitus (eng. pattern matching) on funktionaaliselle ohjelmoinnille tyypillinen piirre, jossa sama funktio voidaan määritellä useita kertoja. Funktiomäärittelyistä vain yhtä sovelletaan tapauskohtaisesti. Modernit funktionaaliset ohjelmointikielet, kuten Haskell, mahdollistavat monimutkaisten tyyppien ja lausekkeiden dekonstruoimisen hahmonsovituksen avulla [3]. Hahmonsovituksen toteutus on käytännössä case-lauseke, jossa lausekkeen ehto kuvaa syötetyn parametrin tietotyyppiä tai sen rakenneta. Tämän avulla ohjelman suoritus jakautuu syötetyn muuttujan tyyppin tai rakenteen mukaan. [1]

Esimerkiksi kertomafunktio voidaan toteuttaa rekursiivisesti hahmonsovituksen avulla. Tässä esimerkissä rekursion alkuarvo määritetään syötteelle 0, ja muut syötteet käsitellään parametrina n .

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

2.4 Funktionaalisen laskentamallin seuraukset

Puhtaasti funktionaalinen laskentamalli karsii ohjelmasta pois tietyt bugityypit. Puhdas sivuvaikutukseton laskenta on automaattisesti säieturvallista (eng. thread safe) tarkoittaen, ettei sitä tarvitse manuaalisesti synkronoida. Koska puhtaan funktion lopputulos riippuu ainoastaan syötteistä, ei syötteiden laskujärjestyksellä ole väliä. Laskennan lopputulosta voidaan siten pitää deterministisenä, mikä helpottaa monisäikeistystä.

Toistettavuus (eng. reproducibility) tarkoittaa, että ohjelman tietynlainen toiminta on toistettavissa. Deterministisyys ja toistettavuus parantavat ohjelman testattavuutta. Jos puhdas funktio toimii tietyllä tavalla kerran, se toimii niin myös joka kerta ajon aikana. Toistettavuus helpottaa ajon aikana löydetyn virheellisen tilan toistamista syöttämällä sama tarkkaan määritetty virheellinen tila testiympäristöön. [7]

Funktionaaliset ohjelmointikielet ovat olleet hitaampia kuin niiden perinteisemmät imperatiiviset sukulaiset, kuten C-kieli, mutta ero on ajan kuluessa kaventunut. Kääntäjäoptimoinneista huolimatta imperatiiviset ohjelmointikielt tulevat aina olemaan optimoidumpia imperatiivisella laitteistolla. Imperatiivisille tietorakenteille on nykyään olemassa monia funktionaalisia vaihtoehtoja, jotka voivat parantaa ohjelman suoritusaikaa ja säästää muistia. [8]

3 Web-kehitys

3.1 World Wide Web

World Wide Web tai WWW on järjestelmä tiedon jakamiseen, joka käyttää Internet-verkkoa. Web-sisältöä luetaan selaimella, joka hakee sisällön web-palvelimelta HTTP-siirtoprotokollan avulla. Verkkosivujen sisällön kuvaamiseen on perinteisesti ollut kolme tekniikkaa: HTML-kieli sivun sisällön kuvaamiseen, CSS-kieli sivun ulkoasun kuvaamiseen ja JavaScript-kieli sivun toiminnallisuuden toteuttamiseen. On myös mahdollista käyttää esimerkiksi Flash- ja Java-liitännäisiä, mutta nykyään on tyyppisempää käyttää pelkästään JavaScript-kieltä. [9]

3.1.1 Staattiset verkkosivut

Alkuperäinen tapa jakaa web-sisältöä on staattisena verkkosivuna (eng. static web-site). Staattinen verkkosivu koostuu HTML-sivuista, dokumenteista ja mediasta, jotka luetaan suoraan web-palvelimen massamuistista, ilman että palvelin tekee niihin muutoksia ajon aikana. Staattiset verkkosivut sisältävät usein HTML-, CSS- ja multimediasisältöä, sekä selaimessa ajettavia skriptejä tai ohjelmia, jotka on kirjoitettu JavaScript-kielellä tai muulla selaimen tulkitsemalla kielellä. [10]

3.1.2 Dynaamiset verkkosivut

Dynaaminen verkkosivu tarkoittaa sivua, joka ei ole suoraan selaimen luettavassa muodossa. Dynaamisen sivun back-end muodostaa selaimella renderöityvän sivun ajon aikana. Dynaamisen verkkosivun operaatioiden toteutus ei olekaan rajattu selaimen tulkittaviin ohjelmointikieliin, vaan käytännössä minkä tahansa kielen käyttäminen on mahdollista. Yleisimpiä kieliä dynaamisen verkkosivun toteuttamiseen ovat PHP, Python, Perl, Ruby, Java, C# ja Node.js. Dynaaminen verkkosivu tallentaa useimmiten sisältönsä tietokantaan. [10]

3.2 Back-end

Back-end tarkoittaa verkkosivun tai -sovelluksen palvelinpäässä ajettavaa osaa, joka ei näy käyttäjälle. Koska back-end-ohjelmaa ajetaan palvelinlaitteistolla, sen toteuttamiseen on paljon enemmän vapautta kuin front-endin, jonka täytyy olla selaimen tulkittavissa. Back-end vastaa usein kaikesta varsinaisesta toiminnallisuudesta verrattuna front-endiin, joka toteuttaa pelkän käyttöliittymän ohjelmalle. Back-endiin kuuluu usein myös tietokanta, johon sovelluksen data tallennetaan. [11]

Back-end kommunikoi front-endin kanssa jonkin yhteisen rajapinnan (eng. Application programming interface, API) kautta. Rajapinta on yleensä pääasiallinen tiedonsiirtokanava back-endin ja front-endin välillä. Front-end lähettää rajapintaan kutsun useimmiten HTTP-protokollan avulla, palvelin vastaanottaa kutsun ja suorittaa kutsutun toimenpiteen. Rajapintaa suunniteltaessa tulee muotoilla rajapinnan rakenne vastaamaan sekä ohjelman datarakennetta, että käyttäjän tai front-endin vaatimuksia. Rajapinta tarjoaa staattiselle front-endille mahdollisuuden dynaamiseen ajonaikana haettavaan sisältöön ja sen muokkaamiseen. Rajapinta voi tarjota front-endille käyttäjän tunnistautumisen (eng. authentication) ja valtuuttamisen (eng. authorization). [11]

3.3 Front-end

Front-end tarkoittaa verkkosivun tai -sovelluksen käyttäjälle näkyvää osaa. Koska front-endiä suoritetaan selaimessa, se on pakko toteuttaa web-yhteensopivilla tekniikoilla, joista yleisimmät ovat HTML, CSS ja JavaScript. JavaScript on nykypäivänä täysiverinen ohjelmointikieli, jolla voidaan toteuttaa interaktiivisuutta sekä manipuloida verkkosovelluksen rakennetta ja ulkoasua. [11]

Front-endin keskeisin tehtävä on käyttöliittymän ja tiedon näyttäminen käyttäjälle. Tästä huolimatta modernissa front-endissä on paljon muitakin asioita joita tulee ottaa huomioon. Front-end ohjelmakoodi muodostaa ja muokkaa verkkosivun HTML DOM-puuuta (eng. DOM tree), joka kuvaa sivun rakenteen. Front-end mahdollistaa myös käyttäjältä tulevien DOM-tapahtumien (eng. DOM event) vastaanottamisen ja käsittelyn. Modernilta front-endiltä odotetaan myös responsiivista designia, joka tarkoittaa että sama sivu toimii usealla erikokoisella päätelaitteella. [12]

Modernissa front-end-kehityksessä on tyypillistä toteuttaa verkkosovelluksen ohjelmakoodi jollain korkeamman tason ohjelmointikiellä, joka muunnetaan web-natiiviksi käännösvaiheessa. Usein käytetään esimerkiksi CSS-tyyliekieleksi kääntyvää SCSS-kieltä. Kääntäjiä, jotka kääntävät front-end-kieliä selainyhteensopiviksi kutsutaan käännöstyökaluiksi (eng. build tool). Esimerkiksi tämän tutkielman käsittelykohteena oleva Elm-kieli ei ole sellaisenaan selainyhteensopiva, vaan vaatii kääntämisen HTML- ja JavaScript-kielille [13]. [11]

3.4 Funktionaalinen web-ohjelmointi

Funktionaalinen web-ohjelmointi on kerännyt suosiota viime vuosina. Funktionaalinen ohjelmointi soveltuu erinomaisesti web-ohjelmointiin sen toistettavuuden ja testattavuuden ansiosta. Bisneslogiikan, tietokantalogiikan ja ohjelmistokehyskoo-

din (eng. framework boilerplate) yksinkertaisempi erottelu on myös perustelu funktionaaliselle web-ohjelmoinnille. Back-end sovelluksen ohjelmointikieleksi voikin valita esimerkiksi Clojure- tai Elixir-kielen. [14] Funktionaalinen ohjelmointi on yleistä etenkin front-end kehityksessä funktionaalisia piirteitä suosivien tekniikoiden, kuten React, ansiosta. Funktionaalinen front-end-ohjelmointi tuo tyypillisiä funktionaalisen ohjelmoinnin hyötyjä, kuten parantaa ohjelman suorituskykyä ja testattavuutta. [5] Koska tutkielman aihe on funktionaalinen front-end web-kehitys, käsitellään seuraavaksi enemmän front-endia.

Funktionaalista paradigmaa toteuttava front-end perustuu datan ja näkymän yhdistämiseen puhtaiden funktioiden avulla. Tämänkaltaisen front-end vaatii jonkin eksplisiittisen mekanismin datan muuttamiseen ja näkymän uudelleen renderöintiin. Funktionaaliset front-end tekniikat suosivatkin usein muuttumatonta (eng. immutable) tilanhallintaa, joka vaatii tilan uudelleenasettamisen sen muuttamiseksi. Tämänkaltaiselle tilanhallinnalle on myös tyypillistä säilöä koko ohjelman tila yhdessä paikassa, ja muuttaa sitä lähettämällä sille käskyjä (eng. dispatching actions). Käskyn vastaanottaa puhdas funktio, joka ottaa parametreina käskyn ja aiemman tilan, palauttaen ohjelman uuden tilan. Näin ohjelma on näennäisesti puhtaan funktionaalinen ja sivuvaikutukseton. [5]

Kuten tosielämän ohjelmille on tyypillistä, front-endissa tarvitaan myös sivuvaikutuksia joidenkin toiminnallisuuksien toteuttamiseen. Sivuvaikutukset ovat hyödyllisiä esimerkiksi tiedon tallentamiseen selaimen muistiin ja satunnaisten numeroiden tuottamiseen. Funktionaaliset front-end ohjelmistokehykset tarjoavatkin tyypillisesti toiminnallisuuden sivuvaikutusten toteuttamiseen funktionaaliselle ohjelmoinnille tyypillisen rajatusti. [13][15]

3.5 Funktionaalinen reaktiivinen ohjelmointi

Reaktiivinen ohjelmointi tarkoittaa asynkronisen datan ja tapahtumien kytkemistä ohjelmointikielen tai ohjelmistokehyksen sisäänrakennetuilla tekniikoilla muuttumattomiin datatyyppeihin. Muuttumattomien datavirtojen käsittely voidaan toteuttaa puhtaasti funktionaalisilla operaatioilla. Tämä on funktionaalisen reaktiivisen ohjelmoinnin (eng. Functional Reactive Programming, FRP) keskeinen konsepti, jota esimerkiksi React ja Elm hyödyntävät [13][15]. [16]

Funktionaalinen reaktiivinen ohjelmointi mahdollistaa perinteisten puhtaiden funktioiden (esim. map, filter, reduce) käyttämisen sovelluksen kuvaamiseen. Keskeinen idea on, että ohjelman ympäristöstä tuleva data sidotaan näennäisesti muuttumattomiin signaaleihin, joita voidaan käsitellä puhtaasti funktionaalisesti. Kun data muuttuu, suoritetaan laskenta uudestaan, uudella arvolla. Toteutus on tällöin tavallaan muuttumaton, vaikkakin data on muuttuvaa. Reaktiiviseen ohjelmointimalliin soveltuvia tapahtumia ja datalähteitä front-end web-sovelluksessa ovat muun muassa DOM-tapahtumat ja rajapinta-kutsut. [13]

4 Funktionaaliset front-end ohjelmistokehykset

Tämän tutkielman tutkimuskohteiksi on valittu funktionaaliset reaktiiviset ohjelmistokehykset React-kirjasto ja Elm-ohjelmointikieli. React-kirjastoa ja Elm-kieltä tutkitaan yleisesti funktionaalisen ohjelmoinnin kannalta, sekä tarkemmin siitä, miten ne toteuttavat tilankäsittelyn, mahdollistavatko ne sivuvaikutukset ja miten toteuttavat asynkroniset tapahtumat.

4.1 Ohjelmistokehysten esittely

React on Facebookin ylläpitämä avoimen lähdekoodin käyttöliittymäkirjasto, jota käytetään tyypillisesti JavaScript-kielen kanssa. React on alunperin julkaistu vuonna 2013 ja se on kirjoitushetkellä vuonna 2020 maailman suosituin front-end ohjelmistokehys noin yhdeksällä miljoonalla viikoittaisella latauksella NPM-palvelusta [17]. React-kirjaston syntaksi on deklaratiiivista, ja se mahdollistaa käyttöliittymän pilkkomisen komponentteihin. React ei ota kantaa muihin tekniikkaratkaisuihin, vaan keskittyy pelkästään yksittäisen DOM-puun hallitsemiseen. Ohjelmistokehittäjälle jää näin täysi valta valita muut teknologiat vapaasti. [15]

Elm on funktionaalinen ohjelmointikieli, joka käännetään JavaScript-kieleksi. Elmin on alunperin kehittänyt Evan Czaplicki osana maisterin tutkielmaansa (eng. Master's thesis) vuonna 2012. Elm ei ole tutkielman kirjoitushetkellä saavuttanut

suurta suosiota, vaan sen NPM-paketinhallinnan latausmäärä on vain noin kolmekymmentä tuhatta latausta viikossa [17]. Elm keskittyy web-pohjaisten graafisten käyttöliittymien deklaratiiiviseen toteuttamiseen. Elm kytkeytyy tavallisen HTML DOM-puun solmuun. Elm ei myöskään ota kantaa muihin tekniikkaratkaisuihin, kuten tyylikieleen, mutta Elm-koodissa on hyvin vaikeaa käyttää muilla kielillä kirjoitettuja komponentteja ja kirjastoja. [13]

4.2 Funktionaalinen ohjelmointi

React korostaa funktionaalista ohjelmointia yli imperatiivisen ohjelmoinnin, mutta sallii silti kaiken tavallisen imperatiivisen JavaScript-syntaksin. React ei pystykään välttämään muuttuvia arvoja tai sivuvaikutuksia, mutta rohkaisee silti ohjelmoidaan funktionaalisesti, muun muassa tarjoamalla tilanhallinnan, joka soveltuu funktionaaliseen ohjelmointiin. [15]

Elm pyrkii olemaan puhtaasti funktionaalinen ohjelmointikieli, eikä salli lainkaan esimerkiksi muuttuvia arvoja tai epäpuhtaita funktioita. Elm ei myöskään salli tilanteita, jotka voisivat aiheuttaa ajonaikaisia virheitä, vaan pakottaa virheiden käsittelyn funktionaalisesti datana. Elm-kielen ajonaikanen ohjelmistokehys abstraktoi näennäisesti kaikki epäpuhtaudet sisällensä, joten Elm-kielillä toteutettu ohjelma on näennäisesti puhtaan funktionaalinen. [13]

4.3 Funktionaalinen tilankäsittely

React tukee lokaalia tilaa missä tahansa komponentissa, joka on ristiriidassa funktionaalisen paradigman kanssa. Tila React-kirjastossa on kuitenkin muuttumaton datarakenne, jonka muuttaminen vaatii sen korvaamisen uudella datalla. Tilan muokkaamiseen on perinteisesti käytetty React-kirjaston Component-luokan `setState`-metodia, joka ylikirjoittaa tilan.

```
// väärin  
  
this.state.greeting = 'Moikku';  
  
// oikein  
  
this.setState({ greeting: 'Moikku' });
```

React-kirjasto tukee nykyään vakiona myös epäpuhtauksia funktioiksi abstraktoivia koukkuja (eng. hooks). Koukut mahdollistavat esimerkiksi tilankäsittelyn puhtaassa funktionaalisessa komponentissa, ilman että tarvitaan luokkaa.

```
const Example = () => {  
  // greeting on tila, joka muuttuu reaktiivisesti  
  // setGreeting on funktio, jonka kutsuminen muuttaa tilan  
  const [greeting, setGreeting] = useState('Moikku');  
  return (  
    <h1>{greeting}</h1>  
  );  
}
```

Funktionaalisen React-sovelluksen toteutukseen on tyypillistä käyttää funktionaaliselle ohjelmalle tyypillistä tilamallia, jossa koko sovelluksen tilaa kuvataan yhdessä paikassa. Tämänkaltaisen tilanhallinta yksinkertaistaa ymmärtämään tilan muutosten vaikutusta sovellukseen. Tyypillinen tapa toteuttaa keskitetty tilanhallinta funktionaalisesti on reducer-funktion avulla, joka palauttaa uuden tilan aiemman tilan ja tapahtuman perusteella. Modernissa React-kirjastossa tämänkaltaista tilanhallintaa voidaan käyttää useReducer-koukun avulla. Vastaavan toteutuksen tarjoaa myös kolmannen osapuolien toteutuksista suosituimpiin kuuluva Redux-kirjasto [14]. Yksinkertaisiin sovelluksiin on silti riittävää käyttää lyhyemmän syntaksin natiiveja vaihtoehtoja, kuten perinteinen setState-toteutus ja useState-koukku. [15]

Elm-kielen arkkitehtuurissa tila on Model-nimisessä abstraktoidussa datamallissa. Model-tilan kuvaaminen ja päivittäminen on hyvin samankaltaista, kuin React-

kirjaston Redux-tilanhallinta. Redux on ottanutkin inspiraatiota Elm-kielen Model-tilanhallinnasta. Model-tilan muokkaaminen tapahtuu kutsumalla update-funktiota message-parametrilla, samankaltaisesti, kuin Redux-kirjaston reducer-funktiossa. [13]

```
-- MODEL

type alias Model = Int

init : Model
init =
    0

-- UPDATE

type MSG = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            model + 1
        Decrement ->
            model - 1
```

Sekä React-kirjaston, että Elm-kielen tilanhallinta on hyvin tyypillistä funktionaalista ohjelmointia. Kumpikaan ohjelmistokehys ei suosittele muuttuvaa tilaa, vaan tilaa säilötään muuttumattomassa tietorakenteessa. React-kirjastossa yksittäisellä komponentilla voi olla lokaali tila, joka ei ole Elm-kielessä mahdollista, vaan kaikki tila on keskitetysti Model-rakenteessa.

4.4 Sivuvaikutukset

// Reactissa: imperatiivinen ohjelmointi, useEffect

// Elmissä: Elm-runtime abstraktoi, innokas laskenta aiheuttaa sivuvaikutuksia

4.5 Funktionaalinen tapahtumankäsittely

// eventit, api-kutsut & asynkronisuus

5 Loppupäätelmät

- Kerrataan havainnot
- Ei uutta asiaa
- Tiivis, 1 sivu

Lähdeluettelo

- [1] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," 1989.
- [2] A. Sabry, "What is a purely functional language?" *Journal of Functional Programming*, 1998.
- [3] M. Lipovaca, "Learn You a Haskell for Great Good!: A Beginner's Guide," 2011.
- [4] F. Cardone ja J. R. Hindley, "History of Lambda-calculus and Combinatory Logic," *Handbook of the History of Logic*, vol. 5, 2006.
- [5] A. Banks ja E. Porcello, "Learning React: Functional Web Development with React and Redux," 2017.
- [6] D. A. Watt, "Programming Language Design Concepts," 2006.
- [7] M. Finifter, A. Mettler, N. Sastry ja D. Wagner, "Verifiable Functional Purity in Java," 2008.
- [8] C. Okasaki, "Purely Functional Data Structures," 1996.
- [9] D. Flanagan, "JavaScript – The definitive guide (6 ed.)," 2016.
- [10] H. Pettersen, "From Static and Dynamic Websites to Static Site Generators," 2016.
- [11] C. Northwood, "The Full Stack Developer," 2018.

- [12] C. Aquino ja T. Gandee, "Front-End Web Development: The Big Nerd Ranch Guide," 2016.
- [13] E. Czaplicki, "Elm website," 2020.
- [14] L. Halvorsen, "Functional Web Development with Elixir, OTP, and Phoenix," 2018.
- [15] Facebook Inc., "React website," 2020.
- [16] Z. Hu, J. Hughes ja M. Wang, "How functional programming mattered," 2015.
- [17] J. Potter, "NPM Trends," 2020.