jamk.fi

Funktionaalisen ohjelmoinnin edut

Case: Mairion

Ville Jokela

Opinnäytetyö
Toukokuu 2018
Tekniikan ja liikenteen ala
Insinööri (AMK), ohjelmistotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Jokela, Ville	Opinnäytetyö, AMK	Toukokuu 2018
	Sivumäärä 41	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Funktionaalisen ohjelmoinnin edut
Case: Mairion

Tutkinto-ohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
Ari Rantala

Toimeksiantaja(t)

Tiivistelmä

Tavoitteena oli selvittää, mitä etuja funktionaalisella ohjelmoinnilla väitetään olevan ja miten nämä edut tulevat ilmi käytännön projektissa. Kyseisenä käytännön projektina toteutettiin REST-rajapinta tehtävien- ja ajanhallintapalvelulle käyttäen funktionaalisen ohjelmoinnin tekniikoita.

Valittiin funktionaalisen ohjelmoinnin väitetyistä eduista tutkittavaksi sopivat: koodin tiiviys, rinnakkais-suoritukseen soveltuvuus ja abstraktiotaso.

REST-rajapinta toteutettiin Scala-ohjelmointikielellä, käyttäen http4s, doobie ja Circe kirjastoja. Ajan puutteen vuoksi toteutus ei täyttänyt kaikkia ohjelmistolle asetettuja vaatimuksia, mutta sitä pystyi silti käyttämään etujen arvioimiseen.

Etujen esiintymistä tutkittiin vertaamalla toteutettua ohjelmistoa opinnäytetyön tekijän aiemmin toteutettuun samankaltaiseen ohjelmistoon, joka oli toteutettu Java-ohjelmointikielellä ja käytti Spring MVC sekä JPA kirjastoja.

Tiiviyttä ei voitu arvioida koska Scala-toteutus erosi Java-toteutuksesta liikaa. Scala-toteutus arvioitiin hieman Java-toteutusta soveliaammaksi rinnakkais-suoritukseen. Scala-toteutus todettiin huomattavasti korkeammalla abstraktiotasolla ohjelmoiduksi.

Abstraktiotason tuloksesta kuitenkin kyseenalaistettiin, että johtuiko se funktionaalisesta ohjelmoinnista. Tuloksista vedettiin johtopäätös, että ohjelmointiparadigman etujen havaitseminen yksittäisessä projektissa on hankalaa ja eri ohjelmointiparadigmoilla toteutettuja ohjelmistoja on hankala verrata.

Avainsanat (asiasanat)
Funktionaalinen ohjelmointi, monadi, REST, Scala

Muut tiedot (salassa pidettävät liitteet)

jamk.fi

Description

<table>
<tr><td rowspan="3">Author(s)
Jokela, Ville</td><td rowspan="2">Type of publication
Bachelor's thesis</td><td>Date
May 2018</td></tr>
<tr><td>Language of publication:
Finnish</td></tr>
<tr><td>Number of pages
41</td><td>Permission for web publi-
cation: x</td></tr>
<tr><td colspan="3">Title of publication
Advantages of Functional Programming
Case: Mairion</td></tr>
<tr><td colspan="3">Degree programme
Software Engineering</td></tr>
<tr><td colspan="3">Supervisor(s)
Rantala, Ari</td></tr>
<tr><td colspan="3">Assigned by</td></tr>
</table>

Abstract

The objective was to find out what advantages functional programming is alleged to have, and observe how these advantages are reflected in a practical project (a REST API for a task and time management web service).

From the alleged advantages, the ones suitable for study were chosen: conciseness of the code, suitability for parallel execution, and level of abstraction.

The REST API was implemented in the Scala programming language, using http4s, doobie, and Circe libraries. Due to lack of time, the implementation did not meet all set requirements; however, it could still be used to evaluate the advantages of functional programming.

The chosen advantages were studied by comparing the Scala implementation with another similar software that was implemented approximately a year earlier using the Java programming language, Spring MVC, and JPA.

Code conciseness could not be evaluated because the Scala implementation ended up too different in comparison to the Java implementation. The Scala implementation was evaluated as slightly more suitable for parallel execution. The Scala implementation was evaluated to have been implemented at a significantly higher level of abstraction.

However, it was questioned whether functional programming was the cause of the higher level of abstraction. Based on the results, it was concluded that observing the advantages of the used programming paradigm is difficult in the context of a single project, and that comparing software implemented with different programming paradigms is also difficult.

Keywords/tags (subjects)
Functional programming, monad, REST, Scala

Miscellaneous (Confidential information)

Sisältö

1 Johdanto ... 5

2 Funktionaalinen ohjelmointi ... 5

2.1 Yleistä ... 5

2.2 Muuttumattomuus ... 6

2.3 Puhtaat funktiot ... 7

2.4 Rekursio ... 9

2.5 Funktiot arvoina ... 10

2.6 Korkeamman asteen funktiot ... 11

2.7 Sivuvaikutusten esittäminen ... 11

2.7.1 Yleistä ... 11

2.7.2 Monadit ... 12

2.7.3 Vapaat monadit ... 13

2.8 Edut ... 14

3 Mairion ... 15

3.1 Tausta ... 15

3.2 Nykyinen järjestelmä ... 15

3.2.1 Tehtävät ... 16

3.2.2 Aika ja tavoitteet ... 18

3.3 Rajaus ... 20

3.4 Vaatimukset ... 21

4 Työn toteutus ... 21

4.1 Etujen arviointi ... 21

4.2 Rajapinnan suunnittelu ... 22

4.2.1 Funktionaalisuus ... 22

4.2.2 Transaktioloki-arkkitehtuuri ... 22

4.3 Määrittely ... 23
4.3.1 Resurssit ... 23
4.3.2 Autentikointi ... 23
4.3.3 Ryhmien käyttöoikeudet ... 24
4.4 Teknologiavalinnat ... 25
4.4.1 Yhteiset kriteerit ... 25
4.4.2 Ohjelmointikieli ... 25
4.4.3 Persistenssi ... 26
4.4.4 HTTP-kirjasto ... 26
4.4.5 Datan siirtoformaatti ... 26
4.5 Rajoitukset ... 27
4.6 Suunnitelma ... 27
4.6.1 Vapaa monadi ... 27
4.6.2 Tietokanta ... 28

5 Tulokset ... 28
5.1 Mairion ... 28
5.1.1 Tilanne opinnäytetyön lopulla ... 28
5.1.2 Käyttöesimerkki ... 29
5.1.3 Erot MairionEE:hen ... 34
5.2 Etujen toteutuminen ... 36
5.2.1 Tiiviys ... 36
5.2.2 Abstraktiustaso ... 36
5.2.3 Rinnakkais-ohjelmointiin soveltuvuus ... 37

6 Pohdinta ... 37
6.1 Etujen arviointi ... 37
6.1.1 Yleensä ... 37

6.1.2 Tiivyys 38

6.1.3 Abstraktiotaso 38

6.1.4 Rinnakkais-ajoon soveltuvuus 38

6.2 Johtopäätökset 39

6.3 Teknologiavalinnat 39

6.4 Funktionaalinen ohjelmointi 39

6.5 Mairion 39

6.6 Yhteenveto 40

Lähteet 41

Kuviot

Kuvio 1. Muistin jakaminen muuttumattomissa tietorakenteissa................................7
Kuvio 2. Näkymä Trellossa olevasta taulusta "Valmistu JAMK:sta" 17
Kuvio 3. Huomisen tehtävät näkymä GQueuesissa 18
Kuvio 4. Kuvakaappaus Togglin työpöytäversiosta 19
Kuvio 5. Viikkosuunnitelman pohja...20
Kuvio 6. Tietueet ja niiden väliset suhteet...28
Kuvio 7. Käyttäjän lisäys 30
Kuvio 8. Työtilan lisäys 31
Kuvio 9. Työtilasta eroaminen...32
Kuvio 10. Työtilan poistoyritys 33
Kuvio 11. Käyttäjän poisto...33
Kuvio 12. Poistetun käyttäjän tietojen lukuyritys 34

Taulukot

Taulukko 1. Yleisessä käytössä olevia monadeja 12
Taulukko 2. Käyttöoikeuden operaatiot ja niiden kohteet 24
Taulukko 3. Esimerkki ryhmän käyttöoikeuksista...25

1 Johdanto

Funktionaalinen ohjelmointi on kasvattanut suosiotaan viime aikoina. Kiinnostusta ovat ajaneet varsinkin JavaScript ja Facebookin avoimen lähdekoodin kirjastot React ja ImmutableJS (After Decades of Neglect, Functional Programming is Finally Going Mainstream. Why Now? 2016) sekä rinnakkaislaskenta, Big Data ja hajautetut järjestelmät (Leonard 2017, 2).

Tämän opinnäytetyön tarkoituksena oli selvittää, mitä etuja funktionaalisella ohjelmoinnilla väitetään olevan sekä tutkia, miten nämä tulevat käytännössä ilmi toteuttamalla ohjelmointiprojekti funktionaalisen ohjelmoinnin tekniikoita käyttäen. Toteutettava projekti on REST-rajapinta tehtävien- ja ajanhallintapalvelulle, koodinimenä Mairion.

Työn toimeksianto on opinnäytetyön tekijän itsensä muotoilema.

2 Funktionaalinen ohjelmointi

2.1 Yleistä

Funktionaalinen ohjelmointi juontaa juurensa matemaattiseen logiikkaan ja Alonzo Churchin lambda-kalkyyliin. Lisp, ensimmäinen funktionaaliseen ohjelmointiin kykenevä ohjelmointikieli kehitettiin jo vuonna 1960. Funktionaalinen ohjelmointi ei koskaan saavuttanut suurta suosiota, vaan sen käyttö rajoittui lähinnä akateemisiin piireihin.

Funktionaalisella ohjelmoinnilla ei ole tarkkaa määritelmää, mutta yksi sen tärkeimmistä käsitteistä on viittauksellinen läpinäkyvyys (referential transparency).

Functional Programming in Scala määrittelee viittauksellisen läpinäkyvyyden seuraavasti: lauseke l on viittauksellisesti läpinäkyvä jos kaikille ohjelmille o, kaikki l:n ilmentymät o:ssa voidaan vaihtaa l:n evaluaation tuloksella ilman että tämä vaikuttaa o:n merkitykseen (Chiusano & Bjarnason 2014, 10).

Tämä tulee esiin kahdella tavalla: muuttumattomat arvot ja puhtaat funktiot.

2.2 Muuttumattomuus

Funktionaalisessa ohjelmoinnissa vältetään muuttujien käyttöä, ja puhtaasti funktionaalisissa kielissä ei ole muuttujia ollenkaan. Jos johonkin arvoon halutaan viitata myöhemmin, se tallennetaan vakioksi. Esimerkiksi Scala sallii sekä muuttujien (var) että vakioiden (val) käytön:

```
var muuttuja: String = "ennen"
muuttuja = "nyt"
// muuttujan arvo on "nyt"
val vakio: String = "hei!"
vakio = "ei onnistu"
// kääntövirhe: vakion uudelleenmäärittely
```

Muuttumattomuus ei rajoitu pelkästään arvoihin, vaan pätee myös olioihin ja tietorakenteisiin. Esimerkiksi Scala käyttää vakiona muuttumattomia tietorakenteita ja sisältää tuen muuttumattomille case-luokille:

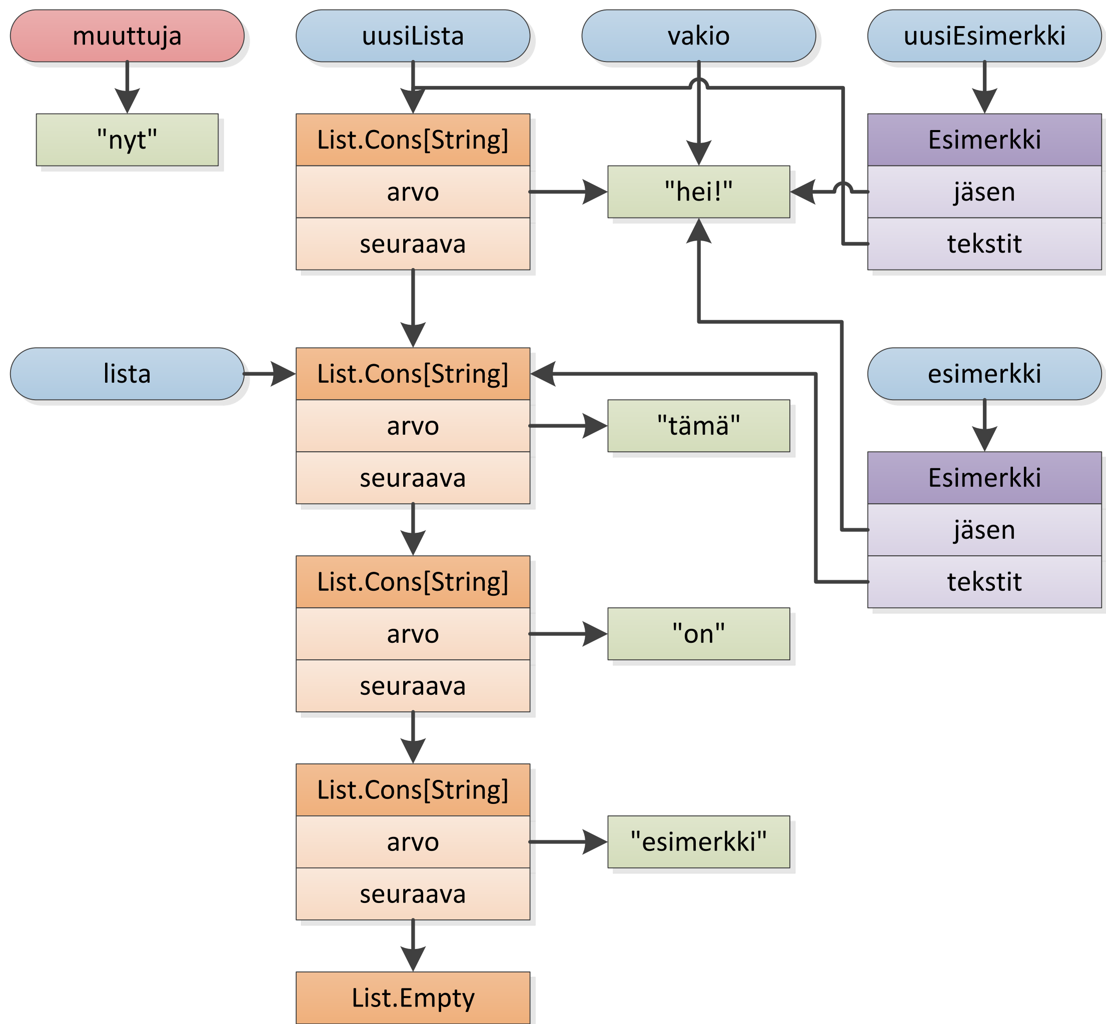
```
val lista: List[String] = List("tämä", "on", "esimerkki")
lista(1) = "on paras"
// kääntövirhe: lista ei tue päivitystä
case class Esimerkki(jäsen: String, tekstit: List[String])
val esimerkki: Esimerkki = Esimerkki(vakio, lista)
esimerkki.jäsen = "testi"
// kääntövirhe: vakion uudelleenmäärittely
```

Jos olioita ja tietorakenteita ei voi muuttaa, miten niitä voi sitten käsitellä? Käsittely tapahtuu luomalla oliosta muokattu kopio tai sisällyttämällä edellinen versio uuteen olioon.

```
val uusiLista = vakio :: lista
// arvo: List("hei!", "tämä", "on", "esimerkki")
val uusiEsimerkki = esimerkki.copy(tekstit = uusiLista)
```

Eikö jatkuva kopiointi vie paljon muistia? Kun sekä tietorakenne että sen sisältämä data on muuttumatonta, siitä ei tarvitse tehdä syvää kopiota, vaan muuttumatonta dataa voi huoletta jakaa tietorakenteiden kesken (Chiusano & Bjarnason 2014, 35).

Kuviossa 1 esitetään aiemmissa esimerkeissä määriteltyjen vakioiden (ja yhden muuttujan) arvot suhteessa toisiinsa.



Kuvio 1. Muistin jakaminen muuttumattomissa tietorakenteissa

Suurissa imperatiivisissa ohjelmissa tehdään usein puolustavaa kopiointia (defensive copying), koska muuten jokin saman ohjelman eri osa saattaisi muuttaa oliota kesken käsittelyä. Funktionaalisessa ohjelmoinnissa tälle ei ole tarvetta (Chiusano & Bjarnason 2014, 10).

2.3 Puhtaat funktiot

Functional Programming in Scala määrittelee asian seuraavasti: funktio f on puhdas jos lauseke f(x) on viittauksellisesti läpinäkyvä kaikille viittauksellisesti läpinäkyville x:ille (Chiusano & Bjarnason 2014, 10).

Viittauksellisesti läpinäkyvän eli puhtaan funktion kutsu voidaan siis korvata sen paluuarvolla ilman, että ohjelman toiminta muuttuu mitenkään, mikä tarkoittaa, että

puhtaan funktion ainoa vaikutuskanava funktion ulkopuoliseen maailmaan on sen paluuarvo. Puhtaan funktion täytyy siis aina palauttaa jokin arvo.

Funktionaalisessa ohjelmoinnissa käytetään termiä "sivuvaikutus", joka tarkoittaa kaikenlaisia toimintoja, jotka käsittelevät muuttuvia asioita funktion ulkopuolella tai tekevät muutoksia ohjelman suoritukseen.

Funktion ei tarvitse edes muuttaa mitään ulkoista rikkoakseen viitteellistä läpinäkyvyyttä. Esimerkkinä `laske`-funktio, joka hakee funktion ulkopuolisesta muuttujasta arvon:

```
def etuliite(teksti: String): String = muuttuja + teksti
val tulos1 = etuliite(" heti")
// arvo: "nyt heti"
muuttuja = "mene"
val tulos2 = etuliite(" heti")
// arvo: "mene heti"
```

Tämä funktio ei voi olla puhdas, koska sen paluuarvo ei ole aina sama, vaikka sitä kutsuttaisiin samalla parametrilla.

Vaikka funktio palauttaisi aina saman arvon vastaanotettuaan samat parametrit, se voi silti rikkoa viitteellistä läpinäkyvyyttä. Esimerkkinä `yhdistä`-funktio, joka yhdistää kaksi tekstipätkää, mutta myös tulostaa nämä konsoliin:

```
def yhdistä(txt1: String, txt2: String): String = {
  println(txt1)
  println(txt2)
  txt1 + txt2
}
val teksti1 = yhdistä("alku", "loppu")
// arvo: "alkuloppu"
val teksti2 = yhdistä("alku", "loppu")
// arvo: "alkuloppu"
```

Jos ohjelmakoodista vaihdettaisiin kaikki `yhdistä`-funktion kutsut niiden paluuarvoilla, ohjelman käytös ja merkitys muuttuisi, koska se ei enää tulostaisi näitä arvoja konsoliin.

Funktiot siis jakautuvat kahteen kastiin: puhtaat funktiot joilla ei ole sivuvaikutuksia ja epäpuhtaat funktiot, joilla on. Puhtaasti funktionaaliset ohjelmointikielet sallivat vain puhtaiden funktioiden käytön.

Esimerkkejä sivuvaikutuksista:

- datan luku tai kirjoitus funktion ulkopuoliseen muuttujaan
- datan luku tai kirjoitus tiedostoon
- datan vastaanottaminen tai lähettäminen verkon yli
- datan luku tai kirjoitus oheislaitteilta (tulostin, skanneri, näyttö, näppäimistö, hiiri jne.)
- ohjelman suorituksen keskeytys
- poikkeuksen aiheuttaminen

2.4 Rekursio

Imperatiivista ohjelmointia oppinut saattaa tässä vaiheessa ihmetellä, miten toistorakenteet toimivat funktionaalisessa ohjelmoinnissa, kun muuttujien käyttö on kiellettyä ja funktiot palauttavat aina saman arvon samoille parametreille, mikä tarkoittaa, että toiston ehtolausekkeen arvo ei voi ikinä muuttua.

Funktionaalisessa ohjelmoinnissa toistorakenteet voidaan usein välttää käyttämällä korkeamman asteen funktioita, mutta niissä tapauksissa missä tämä ei ole järkevää, toisto toteutetaan rekursiolla eli funktiolla, joka kutsuu itseään.

Esimerkkinä tästä on kertoma toteutettuna sekä toistolla että rekursiolla:

```
def kertomaToistolla(n: Int): Int = {
  var jäljellä = n
  var tulos = 0
  while (jäljellä > 0) {
    tulos *= jäljellä
    jäljellä -= 1
  }
  tulos
}
```

```
def kertomaRekursiolla(n: Int): Int = {
  def toistoFunktio(jäljellä: Int, tulos: Int): Int = {
    if (jäljellä > 0)
      toistoFunktio(jäljellä - 1, tulos * jäljellä)
    else
      tulos
  }
  toistoFunktio(n, 0)
}
```

Toiston toteuttamisessa rekursiolla on kuitenkin se ongelma, että jokainen iteraatio lisää ohjelman kutsupinoon (call stack) uuden kehyksen. Ongelman välttämiseksi useimmat funktionaaliset ohjelmointikielet sisältävät häntäkutsu-optimoinnin. Häntäkutsu tarkoittaa tilannetta, missä funktion paluuarvo muodostuu uudesta kutsusta samaan funktioon. Häntäkutsu-optimoinnissa ohjelman kääntäjä muuttaa funktiokutsun hypyksi, jolloin funktio tarvitsee vain yhden kehyksen kutsupinossa.

2.5 Funktiot arvoina

Funktionaaliset ohjelmointikielet käsittelevät funktioita arvoina, jotka voidaan tallentaa vakioon, palauttaa funktiosta tai antaa funktiolle argumenttina yhtä lailla kuin muutkin arvot kuten numerot ja tekstipätkät.

Samankaltaisia ominaisuuksia löytyy muunkin tyyppisistä ohjelmointikielistä, vrt. Runnable-luokan olio Javassa ja funktio-osoitin C-kielessä. Funktionaalisessa ohjelmoinnissa näitä käytetään kuitenkin huomattavasti enemmän, ja tämän takia ne sisältävät tuen anonyymeille eli nimettömille funktioille. Anonyymejä funktioita kutsutaan myös lambda-funktioiksi, nimi tulee lambda-kalkyylistä.

```
def nimetty(numero: Int): Int = {
  numero + 10
}

val anonyymi: Int => Int =
  numero => numero + 10

nimetty(5) == anonyymi(5)
```

2.6 Korkeamman asteen funktiot

Korkeamman asteen funktiot ovat funktioita, jotka antavat paluuarvona funktioita ja/tai ottavat vastaan argumenttina funktioita. Korkeamman asteen funktioiden kanssa käytetään yleensä anonyymejä funktioita.

Funktioiden käsittely arvoina on erittäin tärkeää, koska funktionaaliset ohjelmointikielet käyttävät paljon korkeamman asteen funktioita, varsinkin kokoelma-tietorakenteiden käsittelyssä.

Yleisimmin käytössä oleva korkeamman asteen funktio lienee JavaScriptissä taulukon forEach metodi, joka ottaa parametrina yhden funktion ja suorittaa sen jokaiselle taulukon solulle. Funktionaalisessa ohjelmoinnissa käytetään samanlaista funktiota nimeltä map, joka luo uuden taulukon, jonka solut muodostuvat annetun funktion paluuarvosta, kun funktio suoritetaan siten, että sen parametrina on alkuperäisen taulukon vastaava solu.

```
val numerot = List(1, 2, 3, 4)
val uudetNumerot1 = numerot.map(numero => numero + 10)
// arvo: List(11, 12, 13, 14)
// tämä voidaan esittää myös tiiviillä syntaksilla
val uudetNumerot2 = numerot.map(_ + 10)
// arvo: List(11, 12, 13, 14)
```

2.7 Sivuvaikutusten esittäminen

2.7.1 Yleistä

Miten funktionaalisilla ohjelmointikielillä sitten tehdään hyödyllisiä ohjelmia, jos kaikki sivuvaikutukset ovat kiellettyjä? Puhdas funktio voi vaikuttaa itsensä ulkopuoliseen maailmaan vain palautusarvollaan, mutta paluuarvona voi olla tietorakenne, joka kuvaa epäpuhtaita toimenpiteitä. Puhdasta funktiota kutsunut epäpuhdas koodi voi sitten suorittaa paluuarvon sisältämän tietorakenteen kuvaamat toimenpiteet. Puhtaan funktion paluuarvona voi olla vaikka epäpuhdas funktio.

```
def puhdas(teksti: String): () => Unit = {
  def epäpuhdas(): Unit = {
    println(teksti)
  }
  epäpuhdas
}
// sama lyhyemmin anonyymillä funktiolla
def puhdasTiivis(teksti: String): () => Unit = {
  () => println(teksti)
}
```

Moni funktionaalisesti ohjelmoitu ohjelma muodostuukin puhtaasti funktionaalisesta ytimestä, jonka ympärillä oleva epäpuhdas koodi suorittaa tai tulkitsee puhtaan koodin palauttamia arvoja (Chiusano & Bjarnason 2014, 231). Puhtaasti funktionaalisissa ohjelmointikielissä (joissa koko ohjelma muodostuu puhtaista funktioista) ohjelmointikielen suoritusympäristö hoitaa toimintojen suorituksen.

Sivuvaikutukset voidaan esittää monella eri tavalla, mutta yleisin näistä on monadit.

2.7.2 Monadit

Monadi on kategoriateoriasta peräisin oleva käsite, jota funktionaalisessa ohjelmoinnissa käytetään kuvaamaan ja käsittelemään jonkinlaisessa kontekstissa olevia arvoja.

Tässä yhteydessä keskitytään IO-monadiin, jossa konteksti on että arvon saamiseksi täytyy vaikuttaa ulkopuoliseen maailmaan jotenkin, mutta monadeja käytetään monenlaisten muidenkin kontekstien käsittelyyn (kts. Taulukko 1).

Taulukko 1. Yleisessä käytössä olevia monadeja

Monadin nimi	Monadin konteksti
Future, Promise	Arvo on mahdollisesti saatavilla myöhemmin.
Option, Maybe	Arvoa ei välttämättä ole olemassa.
List	Arvoja saattaa olla monta.
State	Arvo saattaa muuttua.

Vertaa imperatiivisessa ohjelmoinnissa käytettyyn käsitteeseen takaisinkutsufunktio (callback function), jossa ohjelmoijan määrittelemä funktio annetaan kirjastolle tai ohjelmistokehykselle että se voi kutsua sitä tarpeen vaatiessa.

IO-monadi on samankaltainen, mutta monadin sisältämän arvon jatkokäsittely on helppoa. Toisaalta liiallinen jatkokäsittelykutsujen ketjuttaminen saattaa johtaa ongelmiin kutsupinon kanssa.

Esimerkki IO-monadista, joka lukee tiedoston rivit, lisää muistissa olevien rivien alkuun kolme tähteä ja sitten tulostaa nämä konsoliin:

```
def lueTiedosto(nimi: String): IO[Iterator[String]] =
  IO { Source.fromFile(nimi).getLines() }

def kirjoitaRivit(rivit: Iterator[String]): IO[Unit] =
  IO { rivit.foreach(rivi => println(rivi)) }

val tiedostonRivit = lueTiedosto("tiedosto.txt")
val tulostus = tiedostonRivit.flatMap(rivit =>
  kirjoitaRivit(rivit.map(rivi => "*** " + rivi))
)
// Tässä vaiheessa ei ole vielä tehty mitään, vasta
// seuraava rivi suorittaa komennot.
tulostus.unsafeRunSync()
```

IO-monadilla on myös se ongelma että sen sisältämään arvoon tarvittavien toimenpiteiden tarkastelu on mahdotonta, minkä vuoksi toimenpiteiden oikeellisuuden testaaminen täytyy tehdä kuten imperatiivisille ohjelmille.

2.7.3 Vapaat monadit

Vapaa monadi on pohjimmiltaan tapa esittää peräkkäistä laskentaa tietorakenteessa, jotta sitä voidaan tarkastella ja tulkita myöhemmin (De Goes 2015).

Ero normaaliin monadiin on, että vapaa monadi ei sisällä minkäänlaista kuvausta siitä, miten toimenpiteet suoritetaan, vain kuvauksen siitä, mitä toimenpiteitä suoritetaan. Vapaan monadin arvo ei siis itsessään kykene suorittamaan haluttuja toimenpiteitä, vaan tähän tarvitaan tulkki.

Monadin tulkki voi olla joko puhdas tai epäpuhdas. Puhtaalla tulkilla ei ole sivuvaikutuksia, vaan se tulkitsee monadin joksikin toiseksi monadiksi. Kohdemonadi on yleensä konkreettisempi ja epävapaa, jolloin kohdemonadin voi suorittaa ilman toista tulkkia. Epäpuhdas tulkki puolestaan suorittaa vapaan monadin kuvaamat toimenpiteet ilman minkäänlaista välivaihetta.

Funktio, joka palauttaa vapaassa monadissa olevan arvon, antaa sitä kutsuvalle koodille vapauden tulkita vapaan monadin kuvaamat toimenpiteet, miten se parhaaksi näkee, mahdollisesti usealla eri tavalla. Tämä muistuttaa hieman olio-ohjelmoinnissa käytettyjä rajapintoja: vapaan monadin kuvaamat toimenpiteet vastaavat rajapinnan metodeja, vapaan monadin tulkki taas vastaa rajapinnan toteuttavaa luokkaa.

Vapaata monadia voikin käyttää imperatiivisten rajapintojen kuvaukseen. Esimerkiksi Doobie tietokantaohjelmointikirjasto mallintaa JDBC-rajapinnan käyttöä vapailla monadeilla (Norris 2015).

Vapaa monadi toimii samalla myös eräänlaisena riippuvuusinjektiojärjestelmänä. Funktiolla, joka palauttaa vapaassa monadissa olevan arvon, ei ole mitään käsitystä, miten sen palauttamat toimenpiteet suoritetaan.

Vapaan monadin palauttavaa funktiota voi testata tarkastelemalla suoraan sen palauttamaa toimenpidekuvausta, jolloin toimenpiteitä ei tarvitse suorittaa.

Jos vapaan monadin toimenpidekuvaukset on määritelty tarpeeksi abstraktisti, niille voidaan ohjelmoida useita erilaisia tulkkeja. Esimerkiksi jos vapaa monadi kuvaa toimenpiteitä yksinkertaiseen avain-arvo tietokantaan, toimenpiteet voidaan tulkata muun muassa:

- kyselyiksi relaatiotietokantaan
- HTTP-kyselyiksi REST-rajapintaan
- tiedostojärjestelmän luku- ja kirjoitusoperaatioiksi
- sivuvaikutuksettomaksi koodiksi joka pitää avain-arvo parit muistissa, esimerkiksi testien ajamista varten

2.8 Edut

Paul Hudakin (1989, 360) mukaan funktionaalisen ohjelmointi kielten kannattajat väittävät niillä olevan monia etuja, mm. että ohjelmat pystytään kirjoittamaan nopeammin, ovat tiiviimpiä, korkeammalla tasolla, sopivat paremmin muodollisen päätte-

lyn ja analyysin kohteiksi, sekä ovat helpommin ajettavissa rinnakkais-arkkitehtuureilla.

Alvin Alexander (2017) puolestaan kertoo kokeneiden funktionaalisten ohjelmoijien väittävän seuraavia hyötyjä: puhtaiden funktioiden toiminnasta on helpompi järkeillä, testaus ja virheiden etsintä on helpompaa, ohjelmat ovat luodinkestävämpiä ja kirjoitetaan korkeammalla tasolla mikä tekee niistä helpommin ymmärrettäviä, funktioiden signatuurit ovat merkityksekkäämpiä, sekä rinnakkais-ohjelmointi on helpompaa.

Eduista on myös tehty tutkimuksia ja parhaiten tukea löytyy väitteille tiiviydestä (Ray, Posnett, Filkov & Devanbu 2014) ja bugittomuudesta (Nanz & Furia 2015, 6).

3 Mairion

3.1 Tausta

Ajan- ja tehtävienhallinta on taito, jota en aiemmin arvostanut ollenkaan, mutta muutaman viime vuoden aikana mielipiteeni on tehnyt täyskäännöksen ja nykyään olen sitä mieltä, että tämä taito on yksi tärkeimmistä, mitä ihmisellä voi olla.

Ajan- ja tehtävienhallintaan onkin satoja eri tekniikoita, sovelluksia ja verkkopalveluita. Olen kokeillut näistä kymmeniä, kun yritin löytää järjestelmää, joka soveltuisi itselle parhaiten, mutta mikään näistä ei ole ollut täysin tyydyttävä.

Tällä hetkellä käytössäni olevan järjestelmän alkupiste oli neljännesvuosittain asetetut tavoitteet sekä Scrumista lainattu katselmointi ja retrospektiivi. Järjestelmä on kehittynyt hiljalleen, kun olen kokeillut erilaisia tapoja määritellä tehtäviä, asettaa tavoitteita ja seurata ajankäyttöä.

3.2 Nykyinen järjestelmä

Tämänhetkinen järjestelmä, jolla hallinnoin aikaa ja tehtäviä, on saanut vaikutteita mm. Scrumista ja Getting Things Done-järjestelmästä. Varsinaiseen hallinnointiin käytän useaa eri verkkopalvelua, mutta myös käsin piirrettyjä ja kirjoitettuja papereita.

3.2.1 Tehtävät

Minulle luonnollisin tapa käsitellä tehtäviä on hierarkkinen, eli hajotan tehtävät ala-tehtäviin, jotka taas hajotan pienempiin ala-tehtäviin, kunnes ne ovat tarpeeksi pieniä, että ne voi käsittää helposti. Lopputuloksena on puumainen rakenne, jonka syvyys saattaa olla jopa 10 tasoa.

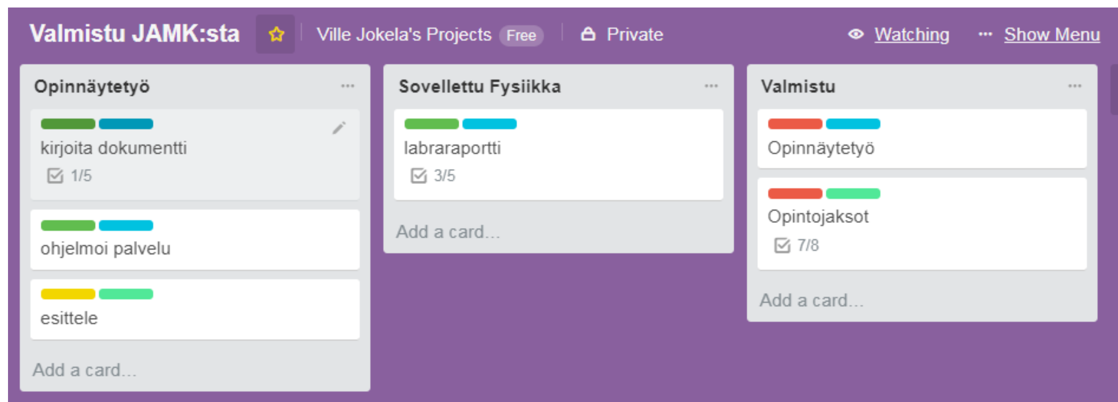
Suurin osa tarjolla olevista sovelluksista ja verkkopalveluista rajoittaa, kuinka syvä tällainen hierarkia voi olla, useimmat sallivat vain kaksi tai kolme tasoa. Monissa ala-tehtäviä ei myöskään voi määritellä samalla tarkkuudella kuin päätehtäviä. Jotkut eivät salli tehtävien jakamista ala-tehtäviin ollenkaan.

Toinen ongelmakohta ovat toistuvat tehtävät. Monissa sovelluksissa toistoasetukset kykenevät toistamaan tehtävän vain joka päivä, viikko, kuukausi tai vuosi. Tämä ei ole riittävä, koska moni tehtävä vaatii toistoa esim. tiettyinä viikonpäivinä, joka toinen viikko tai neljännesvuosittain (eli joka kolmas kuukausi).

Moni tehtävienhallintasovellus kykene näyttämään toistuvista tehtävistä vain seuraavan ajankohdan tehtävän, mikä rajoittaa suunnittelukykyä. Esimerkiksi päivittäinen tehtävä näkyy vain tämän päivän tehtävälistassa, kunnes se merkitään tehdyksi, jonka jälkeen se näkyy vain seuraavan päivän tehtävälistassa.

Ne harvat sovellukset, jotka sallivat tehtäville tarpeeksi syvän hierarkian ja tarpeeksi vapaasti määriteltävän toiston, eivät ole kyenneet yhdistämään näitä kahta, vaan toistuvat hierarkkiset tehtävät aiheuttavat ongelmia.

Tällä hetkellä suurin osa hallinnoimistani tehtävistä sijaitsee Trello-palvelussa, esimerkkinäkymä tästä kuviossa 2. Trellossa tehtävienhallinta tapahtuu luomalla tauluja, luetteloita ja kortteja. Taulu on yksinkertainen elementti, jolla on nimi ja joka sisältää luetteloita. Luettelo on yhtä yksinkertainen, sillä on nimi ja se sisältää kortteja. Kortit ovat Trellon pääasiallinen elementti, ja niille voi lisätä monia yksityiskohtia kuten kuvaus, määräaika, tägejä, tarkistuslistoja, yms. Tarkistuslistaan voi puolestaan lisätä kohtia.

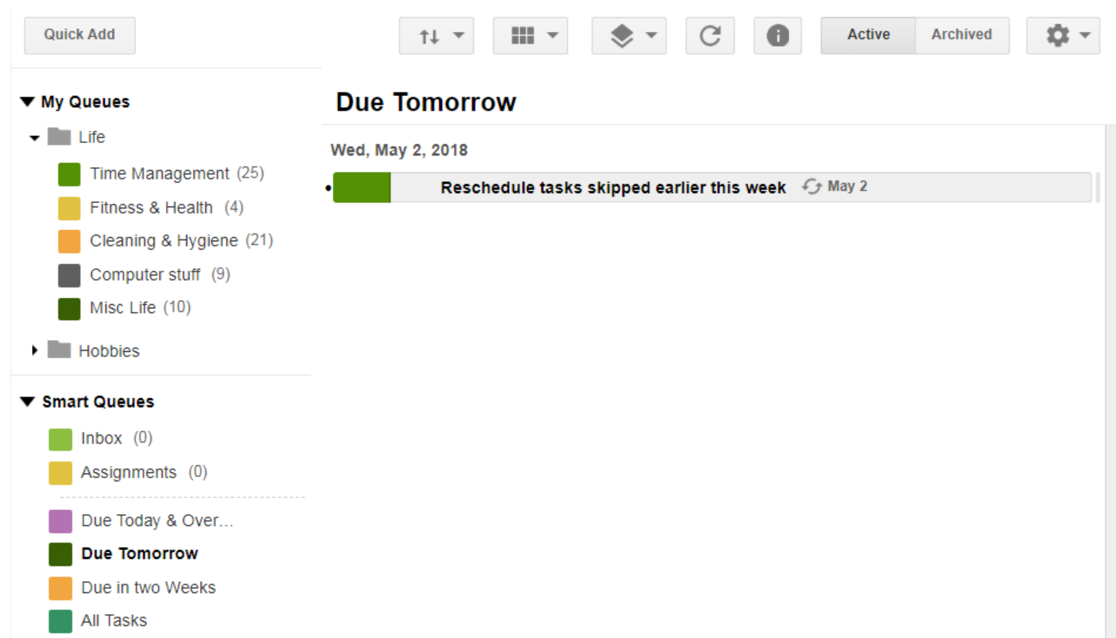


Kuvio 2. Näkymä Trellossa olevasta taulusta "Valmistu JAMK:sta"

Nämä muodostavat viisitasoisen hierarkian, joka onkin moneen käyttöön riittävä, mutta moni tehtävä tarvitsisi vielä syvemmän hierarkian. Käytännössä joudunkin usein tekemään syvemmän tason tehtävistä kortteja, luetteloita tai jopa tauluja, jotta voin kuvata ne tarpeellisella tarkkuudella. Tämän tekniikan käyttäminen kuitenkin hukkaa tehtävien väliset riippuvuussuhteet.

Trellon käytössä on myös muita ongelmia kuten se, että vain korteille voi määritellä tarkempia tietoja ja muilla elementeillä on ainoastaan nimi, sekä liian yksinkertainen toiston määrittely.

Tästä syystä hallinnoinkin toistuvia tehtäviä GQueues-palvelussa, josta esimerkki näkymä kuviossa 3. GQueues tukee sekä hierarkkisia tehtäviä, että monipuolisia toistoasetuksia, mutta nämä eivät toimi yhdessä. Yksi ratkaisu tälle on hajottaa hierarkkinen tehtävä erillisiksi tehtäviksi ja asettaa kaikille samat toistoasetukset, mutta tämän tekemällä tehtävät menettävät niiden väliset riippuvuussuhteet, eikä tehtävien järjestys ole aina sama jokaisella toistokerralla.



Kuvio 3. Huomisen tehtävät näkymä GQueuesissa

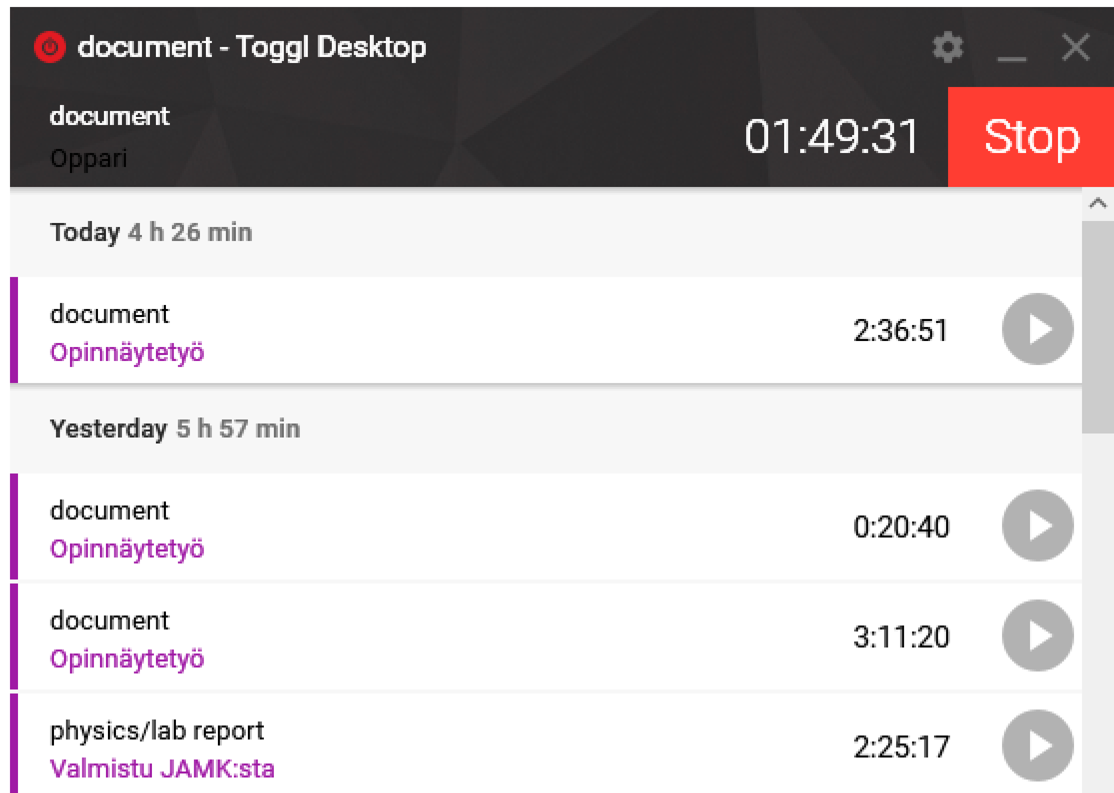
3.2.2 Aika ja tavoitteet

Neljännesvuosittainen tavoitteiden asettaminen on kehittynyt monitasoiseksi, jossa asetan tavoitteita ajanjaksoille, jotka sisältävät pienempiä ajanjaksoja, jotka taas sisältävät pienempiä ajanjaksoja jne. Yhteensä käytän kuutta eripituista ajanjaksoa: vuosi, neljännesvuosi, kuukausi, viikko, päivä, vuorokauden osa. Syy tähän hienojakoisuuteen on taipumukseni viivyttää tehtävän toteuttamista mahdollisimman pitkään. Parhaaksi kokemani tekniikka ongelman välttämiseen on asettaa itselleni tavoittet siten, että viivyttämiselle ei jää paljon tilaa, esimerkiksi tavoitteena saattaa olla kirjoittaa opinnäytetyötä kaksi tuntia aamupäivällä.

Alun perin tavoitteita asettaessa ja katselmoidessa kiinnitin huomiota vain siihen sainko tehtävän valmiiksi kyseisellä ajanjaksolla. Nykyään käytän valmiuskriteeriä vain tehtäville, joiden arvioin tarvitsevan alle kaksi tuntia valmistumiseen. Suuremmille tehtäville asetan tavoitteet käytetyn ajan pohjalta, eli lasken tavoitteen saavutetuksi, jos olen työskennellyt tehtävän parissa vähintään asetetun määrän aikaa tai sain tehtävän valmiiksi.

Ajankäyttöä seuraan Toggl-palvelulla, joka sallii vain kaksitasoisen hierarkian. Ajankäytön merkintään voi asettaa projektin ja tehtävän, mikä aiheuttaa

yhteensopivuusongelmia muun järjestelmän kanssa. Kuvio 4 sisältää esimerkin kahdesta eri tavasta jolla sovitan syvemmästi hierarkkisia tehtäviä tähän malliin: opinnäytetyön olen irroittanut erilliseksi projektiksi "Valmistu JAMK:sta" projektista, kun taas fysiikan opintojakson laboratorioraportin kirjoittaminen sisällyttää hierarkian tehtävä-kenttään.



Kuvio 4. Kuvakaappaus Togglin työpöytäversiosta

Käytän Togglia lähinnä sellaisten tehtävien seurantaan joiden suorisuksessa kestää vähintään muutama päivä. Lyhyempiin tehtäviin käyttämääni aikaa en mittaa tarkasti vaan merkitsen vain arvion.

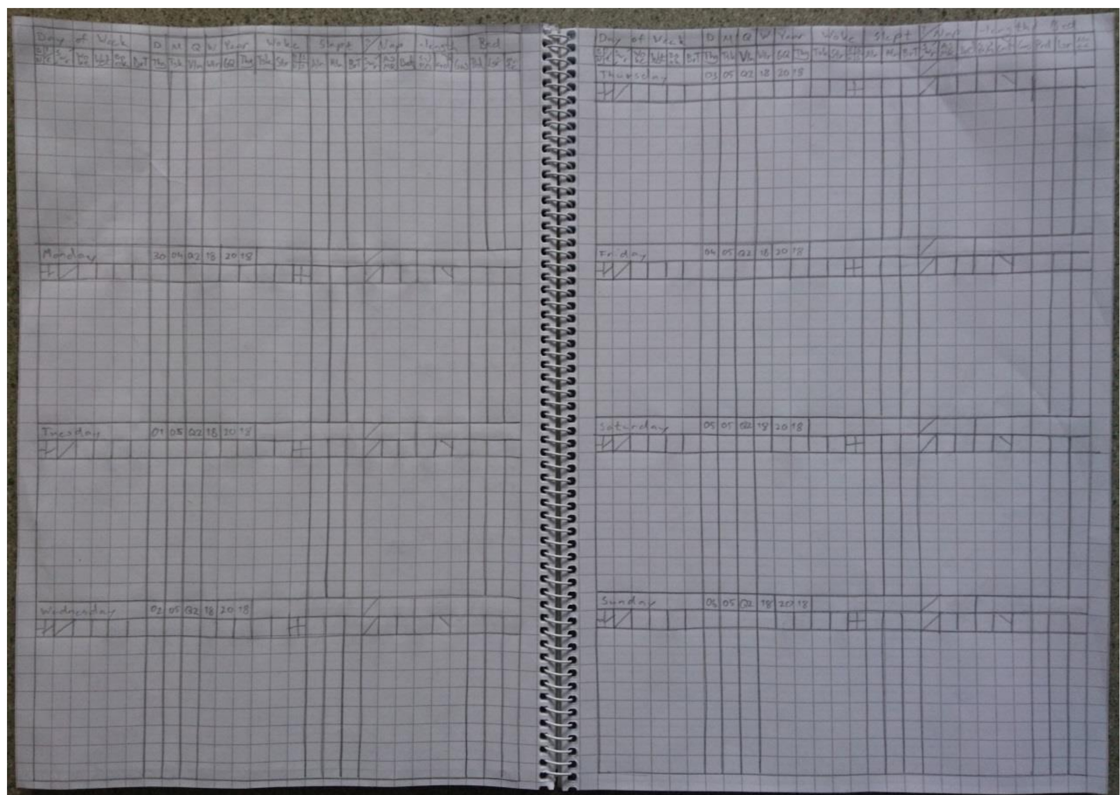
Togglia voi käyttää työpöytä- ja mobiilisovelluksella, sekä internetsivustolla. Itsellä suurimmassa käytössä on työpöytäsovellus koska nämä pitempikestoiset tehtävät ovat yleensä tietokoneella suoritettavia.

Tavoitteiden seurantaan en ole löytänyt sopivaa sovellusta, joten käytän kynää ja paperia. Seurannan voisi hoitaa esimerkiksi taulukkolaskentaohjelmalla, mutta paperin käytöllä on muutama etu tähän verrattuna: ei tarvetta tietokoneelle tai mobiililaitteelle ja helppo soveltaa muuttuviin tarpeisiin sekä epätavallisiin

tilanteisiin. Sillä on myös huonot puolensa: viikkoaukeaman piirtäminen vie aikaa, kirjattujen tehtävien järjestystä on hankala muuttaa ja pitkäkestoisten tehtävien eteneminen ei ole helposti nähtävissä.

Tavoitteiden paperilla seuraamisen huonot puolet ovatkin pääsyy siihen, miksi haluan kehittää oman sovelluksen.

Tärkein osa tavoitteiden seurannassa on viikkosuunnitelma, josta valokuva kuviossa 5. Tämä sisältää tilan koko viikon tavoitteille sekä oman tilan jokaiselle päivälle. Lisäksi aukeamalla on ruudukot päivittäisille tehtäville.



Kuvio 5. Viikkosuunnitelman pohja

3.3 Rajaus

Kehitetyn järjestelmän ei tarvitse toteuttaa kaikkia tämänhetkisen järjestelmän toimintoja, vaan riittää että se vähentää ajan- ja tehtävienhallintaan käyttämääni aikaa. Tällä hetkellä suurin osa hallintaan käytetystä ajasta menee tehtävien seuraamiseen vihkossa, joten jos tämän saa automatisoitua niin sovellus on jo hyödyllinen.

Huomattavin poisjätetty kokonaisuus on toistuvat tehtävät. Syy tämän pois jättämiseen on se, että tätä ei ole missään sovelluksessa toteutettu tyydyttävästi, joten sovelluksen ei tarvitse sisältää tätä ominaisuutta ollakseen vähintään yhtä hyvä kuin kilpailevat tuotteet.

3.4 Vaatimukset

Tärkeimmät ominaisuudet ovat siis tuki hierarkkisille tehtäville, monitasoisille ajanjaksoille, aika- sekä valmius-pohjaisille tavoitteille, sekä käytetyn ajan seurannalle. Koska sovellus on tarkoitus kehittää verkkopalveluna, lisäksi tarvitaan käyttäjien hallinta, työtilat ja toisten käyttäjien kutsuminen yhteisiin työtiloihin, sekä tietoturvaominaisuuksia kuten käyttäjäryhmät ja näiden oikeudet.

Mairionille on suunniteltu toteutettavaksi käyttöliittymiksi ainakin internet-sivusto sekä Android-sovellus.

4 Työn toteutus

4.1 Etujen arviointi

Kuinka funktionaalisen ohjelmoinnin etuja voidaan arvioida yksittäisen projektin rajoissa? Ensimmäisenä täytyy todeta että tulokset tulevat pakosta olemaan jokseenkin subjektiivisia, koska kyse on yhdestä yksin ohjelmoidusta projektista.

Jotta arviointi ei olisi ihan täysin arvailua, käytin vertailupisteenä vuosi sitten Javalla toteuttamaani aiempaa versiota samasta ideasta. Java-toteutus toimi projektityönä JavaEE opintojaksolle ja se toteutettiin käyttämällä Spring MVC:tä ja JPA:ta. Viittaan tähän aiempaan versioon vastaisuudessa termillä MairionEE.

MairionEE käyttää joitain Java 8:n funktionaalisesta ohjelmoinnista vaikutteita ottaneita ominaisuuksia kuten Optional, Stream ja lambda-funktiot, joten vertailupisteenä se ei ole täysin imperatiivinen. Se oli kuitenkin aiemmin toteuttamistani projekteista selvästi samankaltaisin Mairionin kanssa, joten se oli järkevin vertauskohde.

Arvioin funktionaalisen ohjelmoinnin etuja vertaamalla Mairionin ja MairionEE:n ohjelmakoodia seuraavissa kategorioissa:

1. tiiviys
2. abstraktiustaso
3. rinnakkais-ohjelmointiin soveltuvuus

Testattavuus olisi ollut hyvä kategoria arvioitavaksi, mutta MairionEE ei sisällä minkäänlaista testausta, joten tätä vertailua ei voi tehdä.

Virheiden etsimisen helppous ja ohjelmointinopeus täytyi myös jättää pois, koska MairionEE:n toteutuksesta oli tätä työtä tehdessä kulunut jo noin vuosi, joten näistä kummastakaan ei ollut enää tarkkoja muistikuvia.

4.2 Rajapinnan suunnittelu

4.2.1 Funktionaalisuus

Kaikki tämä funktionaalisen ohjelmoinnin opiskelu aiheutti sen että aloin kyseenalaistamaan imperatiivista suunnittelua kaikkialla missä tulin sen kanssa tekemisiin. Suunniteltava REST-rajapinta ei säästynyt tältä käsittelyltä.

Kuinka REST-rajapinnasta voisi tehdä funktionaalisen tai ainakin vähemmän imperatiivisen? Puhtaasti funktionaalinen rajapinta palauttaisi aina saman vastauksen samalle kyselylle.

Tämä voitaisiin toteuttaa sisällyttämällä kyselyyn aikaleima. Tällä tekniikalla saataisiin samalla tietokannan aiemmat arvot nähtäville, mutta samalla se vaikeuttaisi välimuistin käyttöä.

4.2.2 Transaktioloki-arkkitehtuuri

Jonkinlainen listaus muutoksista täytyisi kuitenkin toteuttaa jossain vaiheessa jos palveluun haluttaisiin lisätä kumoa ja tee uudelleen toiminnallisuudet tai tietueiden historian näyttö. Miksi ei siis rakentaa koko rajapinta tällä arkkitehtuurilla?

Päätin että rajapinta ei tarjoa kykyä työtilan sisältämien elementtien suoraan katseluun, luomiseen, muokkaukseen tai poistoon, vaan kaikki tämä tapahtuu epäsuorasti

transaktioilla. REST-rajapinta esittää infrastruktuuriresurssit imperatiiviseen tyyliin kuntaas työtilan sisältämät resurssit esitetään transaktiolokin muodossa.

Voidakseen näyttää työtilan sisältöä, asiakasohjelman täytyy hakea koko transaktiohistoria ja tämän pohjalta rakentaa paikallinen kopio työtilasta. Tämä saattaa vaikuttaa suurelta määrältä työtä yhden tietueen näyttämiseksi ja sitä se olisikin jos yksittäisen tietueen näyttäminen vain kerran samalle henkilölle olisi tyypillinen käyttötapaus, mutta kyseessä on tehtävienhallintasovellus jossa tämä on hyvin harvinaista.

Tehtävienhallintasovelluksen käyttö on pitkäjänteistä ja sisältää paljon hitaasti muuttuvia tietorakenteita.

Transaktioloki-arkkitehtuuri vaikuttaa vahvasti tiedonsiirron määrään. Käyttäjän lisääminen työtilaan vaatii koko transaktiohistorian siirron, mutta tämä on verrattain harvinainen tapahtuma. Tällä arkkitehtuurilla ehdottomasti suurin osa rajapintaan tehdyistä kyselyistä ovat muotoa "Hae kaikki työtilaan X transaktion Y jälkeen tehdyt muutokset" ja palvelun yleisin vastaus tähän on tyhjä lista.

4.3 Määrittely

4.3.1 Resurssit

Resurssit jakautuvat kahteen kategoriaan: suoran käsittelyn resurssit ja transaktiolokin alaisuudessa olevat resurssit. Suoraan käsiteltävissä olevat resurssit ovat: työtila, käyttäjä, ryhmä, lupa, kutsu ja transaktio. Transaktiolokin alaisuudessa on seuraavat resurssit: tehtävä, ajanjakso, tavoite ja ajankäytön merkintä.

4.3.2 Autentikointi

Palvelu hyötyisi suuresti siitä jos se tarjoaisi avoimen rajapinnan jolla se voitaisiin integroida muihin verkkopalveluihin, kuten esimerkiksi Slack. Tästä syystä olisi paras jos käyttäjät kykenisivät määrittelemään kolmannen osapuolen asiakassovelluksille eritasoiset valtuudet heidän tilinsä käyttöön.

OAuth2 olisi juuri sopiva tällaiseen käyttöön, mutta valitettavasti tämä standardi on varsin monimutkainen eikä sen käytön sisällyttäminen ole järkevää aikapaineiden takia.

Autentikointi toteutetaan HTTP Basic Auth:lla.

4.3.3 Ryhmien käyttöoikeudet

Käyttäjäryhmät ja käyttöoikeudet eivät olisi ihan välttämättömiä ensimmäiseen versioon, mutta tietoturvaominaisuudet on parempi rakentaa sisään alkuperäiseen suunnitelmaan kuin yrittää lisätä valmiin ohjelman päälle.

Käyttäjäryhmällä on lista käyttöoikeuksista. Jokainen työtilassa oleva käyttäjä kuuluu vähintään yhteen käyttäjäryhmään ja käyttäjän oikeudet ovat sallivin yhdistelmä kaikista käyttäjän ryhmien oikeuksista.

Käyttöoikeus muodostuu kolmesta osasta: operaatio, oikeuden laajuus ja kohderesurssin tyyppi.

Operaatioita on viisi, mutta transaktioloki-arkkitehtuurin vuoksi kaikki operaatiot eivät ole järkeviä transaktiolokin alaisuudessa oleviin resursseihin (kts. taulukko 2). Näistä operaatioista käytetään lyhennettä BREAD, joka tulee operaatioiden englanninkielisten nimien alkukirjaimista.

Taulukko 2. Käyttöoikeuden operaatiot ja niiden kohteet

Operaatio	Englanniksi	Kohderesurssin kategoria
selaus	browse	suoraan käsiteltävä
luku	read	suoraan käsiteltävä
muokkaus	edit	kaikki
lisäys	add	kaikki
poisto	delete	kaikki

Käyttöoikeuden kohderesurssin tyyppi voi olla mikä tahansa järjestelmän hallinnoima resurssi (kts. luku 4.3.1).

Oikeuden laajuus määrittää sen päteekö käyttöoikeus kaikkiin työtilassa oleviin resursseihin vai pelkästään käyttäjän omistamiin resursseihin.

Peruskäyttäjän käyttöoikeudet voisivat esimerkiksi rajoittua pelkästään transaktioihin ja tehtäviin (kts. taulukko 3).

Taulukko 3. Esimerkki ryhmän käyttöoikeuksista

Operaatio	Laajuus	Kohderesurssin tyyppi
selaus	työtila	transaktio
luku	työtila	transaktio
lisäys	oma	transaktio
lisäys	oma	tehtävä
muokkaus	oma	tehtävä
poisto	oma	tehtävä

4.4 Teknologiavalinnat

4.4.1 Yhteiset kriteerit

Tärkeimmät teknologiavalintoja rajoittavat tosiasiat olivat opinnäytetyön keskittyminen funktionaaliseen ohjelmointiin ja projektiin käytettävän ajan rajallisuus. Ajan niukkuus pakotti valitsemaan teknologioita, jotka olivat syystä tai toisesta nopea oppia. Näistä muodostui lista kriteerejä, jotka pätivät kaikkiin valintoihin:

- mahdollisimman puhtaasti funktionaalinen
- yksinkertainen
- hyvä dokumentaatio ja esimerkit
- aiempi osaaminen soveltuvaa

4.4.2 Ohjelmointikieli

Ainoat ohjelmointikielet joissa minulla oli minkäänlaista kokemusta funktionaalisesta ohjelmoinnista, olivat Scala ja JavaScript. Kummassakin tämä kokemus oli rajoittunut funktionaalis-tyyliseen imperatiiviseen ohjelmointiin, jossa mahdollisimman suuri osa ohjelmoiduista funktioista on puhtaasti funktionaalisia, mutta epäpuhtaita funktioita käytetään myös eikä sivuvaikutuksia eristetä monadeihin.

Vaikka kumpikin kieli soveltuu REST-rajapinnan toteuttamiseen, vain Scalasta löytyi kirjastoja tämän toteuttamiseen puhtaan funktionaalisesti. Lisäksi Scalan standardi-kirjasto ja kolmannen osapuolen kirjastot ovat enemmän suuntautuneet funktionaa-liseen ohjelmointiin, sekä itseltä löytyi enemmän kokemusta Scalan kuin JavaScriptin käytöstä. Ohjelmointikieleksi valikoitui siis Scala.

Scala on Java virtuaalikoneella ajettava hybridi-kieli joka tukee sekä funktionaalista että olio-ohjelmointia. Se ei ole puhtaasti funktionaalinen kieli (epäpuhtaat funktiot ovat sallittuja ja kieli tukee muuttujia), mutta sille on kirjastoja puhtaasti funktionaa-lista ohjelmointia varten.

Vaikka Scalassa pystyy käyttämään mitä tahansa Java-kirjastoa, nämä ovat yleensä suunniteltu imperatiivista ohjelmointia varten. Jotta tuotettu sovellus pysyisi mah-dollisimman puhtaasti funktionaalisena, päätin rajata valinnat Scala-kirjastoihin.

4.4.3 Persistenssi

Aikaisempi kokemus rajoitti hyväksyttävät ratkaisut relaatiotietokantoihin, sekä pa-kotti hylkäämään monimutkaisemmat rajapinnat kuten Slick ja Quill joissa tietokan-nan käsittely tapahtuu täsmäkieltä (domain specific language) käyttämällä. Jäljelle jäi SQL kyselyiden käsin kirjoittaminen ja ainoa puhtaasti funktionaalinen kirjasto tähän tarkoitukseen oli aiemmin tässä työssä mainittu Doobie.

4.4.4 HTTP-kirjasto

Puhtaasti funktionaalisia REST-rajapinnan toteuttamiseen soveltuvia kirjastoja löytyi kaksi, http4s ja Finch. Lopulliseksi valinnaksi muodostui http4s lähinnä paremman dokumentaation ja esimerkkien paljouden vuoksi.

4.4.5 Datan siirtoformaatti

REST-rajapinnan siirtoformaattina käytetään useimmiten JSON:ia, eikä tässä projek-tissa ole mitään syytä poiketa tästä.

JSON:n käsittelyyn käytettävän kirjaston valinta on myös helppo. Circe käyttää samaa funktionaalisen ohjelmoinnin kirjastoa kuin Doobie ja http4s suosittelee sen käyttöä.

4.5 Rajoitukset

Http4s asettaa tiettyjä rajoituksia arkkitehtuurille. Palvelu tulee määritellä yhtenä tai useampana funktiona joka ottaa parametrina pyyntöolion ja palauttaa vastausolion. Suositeltu tapa käsitellä pyyntöoliota on hahmonsovituksella purkaa pyyntö metodiin ja polkuun.

Näitä funktioita voi käsitellä väliohjelmistokomponentilla ja todennus suositellaan toteuttamaan tällä tekniikalla.

Tietokannan käyttöön http4s ei sisällä erillistä tukea joten sen suhteen ei ole mitään rajoituksia.

4.6 Suunnitelma

4.6.1 Vapaa monadi

Rajapinnan pääasiallinen tehtävä on tallentaa ja hakea dataa tietokannasta, mikä pakottaa sivuvaikutuksien käyttöön.

Suunnitelmana oli esittää tietojen haku ja tallennus vapaalla monadilla sekä ohjelmoida tälle kaksi tulkkia, yksi tuotantokäyttöön ja toinen testaukseen. Tuotantokäyttöön tuleva tulkki käyttää Doobie-kirjastoa tietokannan kanssa keskusteluun, kun taas testauskäyttöön tuleva tulkki käyttää vain muistissa olevia yksinkertaisia tietorakenteita tiedon tallentamiseen ja hakemiseen.

Nimenä tällä monadilla on MIO (Mairion Input/Output) ja kontekstina se, että arvojen käsittely vaatii kommunikointia tietovaraston kanssa, mikä puolestaan vaatii sivuvaikutusten käyttöä.

Monadi tarjoaa seuraavat operaatiot:

- käyttäjän todennus
- pyynnön valtuuksien tarkistus
- resurssien selaus
- resurssin luku
- resurssin lisäys
- resurssin muokkaus
- resurssin poisto
- puhtaan arvon nostaminen kontekstiin

4.6.2 Tietokanta

Tietokantaan tallennettavat tietueet ja niiden väliset suhteet on esitetty kuviossa

Invite	
PK	inviteId
FK1,U1 U1	workspaceId email status created expires

Workspace	
PK	workspaceId
U1	name description

WorkspaceMembership	
PK	wsMemberId
FK1,U1 FK2,U1 U1	workspaceId userId joined exited exitType exitNote

User	
PK	userId
U1 U2	email name hashedPassword

InvitedToGroup	
PK,FK1 PK,FK2	inviteId groupId

Group	
PK	groupId
FK1,U1 U1	workspaceId name

GroupMembership	
PK,FK2 PK,FK1	userId groupId

Transaction	
PK	transactionId
FK1 FK2	workspaceId authorId commitTime

Permission	
PK	permissionId
FK1,U1 U1 U1 U1	groupId operationType targetOwner targetResourceType

WorkspaceOperation	
PK	operationId
FK1	transactionId executionTime operationType resourceType resourceId resourceDelta

Kuvio 6. Tietueet ja niiden väliset suhteet

5 Tulokset

5.1 Mairion

5.1.1 Tilanne opinnäytetyön lopulla

Ajan puutteen vuoksi rajapinnan toteutus jäi huomattavasti kesken, mutta suurin osa komponenteista saatiin kuitenkin toimintakuntoisiksi ja rajapinta pystyy käsittelemään rajatun määrän pyyntöjä oikein.

Toimintakuntoisiksi saatiin seuraavat ominaisuudet:

- todennus
- valtuuksien tarkistus
- JSON-muunnokset
- vapaa monadi MIO
    - testaukseen tarkoitettu tulkki

- kaikille resursseille yhteiset toiminnallisuudet
- täydet toiminnallisuudet seuraaville resursseille:
  - käyttäjä
  - työtila
  - työtilan jäsenyys

Puuttumaan jäivät seuraavat ominaisuudet:

- tuotantokäyttöön tarkoitettu MIO-tulkki
- toiminnallisuudet seuraaville resursseille:
  - ryhmä
  - ryhmän jäsenyys
  - lupa
  - kutsu
  - transaktio ja kaikki tämän alaisuudessa olevat resurssit

5.1.2 Käyttöesimerkki

Koska rajapinnalle ei ole vielä toteutettu käyttöliittymää, sen käyttöä täytyy esitellä REST-rajapintojen testaamiseen tarkoitetulla työkalulla.

Kuviossa 7 lisätään käyttäjä, kuviossa 8 luodaan työtila, kuviossa 9 erotaan työtilasta, kuviossa 10 yritetään poistaa työtila jonka jäsenenä käyttäjä ei enää ole, kuviossa 11 poistetaan käyttäjä järjestelmästä ja kuviossa 12 yritetään lukea poistetun käyttäjän tiedot.

add user

Save

METHOD

POST

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/users/

Send

length: 28 bytes

QUERY PARAMETERS

+ Add query parameter

HEADERS

Form

Content-Ty : application/json

+ Add header

Add authorization

BODY

Text

```
1 {
2   "name": "test",
3   "password": "test",
4   "email": "email@email.com"
5 }
```

Text | JSON | XML | HTML

Enable body evaluation

length: 72 bytes

Response

Cache Detected - Elapsed Time: 446ms

201 Created

HEADERS

pretty

```
Content-Type: application/json
Date:         Sat, 19 May 2018 21:07:45 GMT
Content-Leng… 85 bytes
```

COMPLETE REQUEST HEADERS

BODY

pretty


```
{
  1: {
    name: "test",
    password: null,
    hashedPassword: null,
    email: "email@email.com"
  }
}
```


lines nums

length: 85 bytes

Kuvio 7. Käyttäjän lisäys

add workspace

Save

METHOD

POST

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/workspaces/

Send

length: 33 bytes

QUERY PARAMETERS

+ Add query parameter

HEADERS | Form

☑	Authorizat	Basic dGVzdDp0ZXN0
☑	Content-Ty	application/json

+ Add header | Add authorization

BODY | Text

```
1 {
2   "name": "My Workspace",
3   "description": "for real work"
4 }
```

Text | JSON | XML | HTML | ☑ Enable body evaluation | length: 62 bytes

Response

Cache Detected - Elapsed Time: 67ms

201 Created

HEADERS | pretty

```
Content-Type: application/json
Date:         Sat, 19 May 2018 21:16:11 GMT
Content-Leng… 59 bytes
```

COMPLETE REQUEST HEADERS

BODY | pretty

```
{
  1: {
    name: "My Workspace",
    description: "for real work"
  }
}
```

lines nums | length: 59 bytes

Kuvio 8. Työtilan lisäys

add resignation

Save

METHOD

POST

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/workspaces/1/memberships/resignations

Send

length: 59 bytes

QUERY PARAMETERS

HEADERS

Form

Authorizat : Basic dGVzdDp0ZXN0

Content-Ty : application/json

+ Add header

Add authorization

BODY

Text


```
1 {
2   "wsId": 1,
3   "note": "I've got better things to do"
4 }
```


Text | JSON | XML | HTML Enable body evaluation length: 57 bytes

Response

Cache Detected - Elapsed Time: 57ms

201 Created

HEADERS

pretty

```
Content-Type: application/json
Date:         Sat, 19 May 2018 21:17:31 GMT
Content-Leng… 192 bytes
```

COMPLETE REQUEST HEADERS

BODY

pretty


```
{
  1: {
    wsId: 1,
    userId: 1,
    isWorkspaceOwner: true,
    joined: "2018-05-19T21:16:11.781Z",
    exited: "2018-05-19T21:17:31.837Z",
    exitType: {
      Resigned: {}
    },
    exitNote: "I've got better things to do"
  }
}
```


lines nums

length: 192 bytes

Kuvio 9. Työtilasta eroaminen

delete workspace

Save

METHOD SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

DELETE http://localhost:8080/workspaces/1 Send

length: 34 bytes

QUERY PARAMETERS

+ Add query parameter

HEADERS Form BODY

Authorizat : Basic dGVzdDp0ZXN0

XHR does not allow payloads for DELETE request.

+ Add header Add authorization

Response Cache Detected - Elapsed Time: 32ms

403 Forbidden

HEADERS pretty BODY

Date: Sat, 19 May 2018 21:18:30 GMT
Content-Leng… 0 byte

NO CONTENT

COMPLETE REQUEST HEADERS

Kuvio 10. Työtilan poistoyritys

delete self

Save

METHOD SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

DELETE http://localhost:8080/users/self Send

length: 32 bytes

QUERY PARAMETERS

+ Add query parameter

HEADERS Form BODY

Authorizat : Basic dGVzdDp0ZXN0

XHR does not allow payloads for DELETE request.

+ Add header Add authorization

Response Cache Detected - Elapsed Time: 24ms

200 OK

HEADERS pretty BODY pretty

Content-Type: application/json
Date: Sat, 19 May 2018 21:19:07 GMT
Content-Leng… 2 bytes

{}

COMPLETE REQUEST HEADERS

lines nums length: 2 bytes

Kuvio 11. Käyttäjän poisto

Kuvio 12. Poistetun käyttäjän tietojen lukuyritys

5.1.3 Erot MairionEE:hen

Ohjelmakoodin vertailu oli odotettua hankalampaa, koska Mairion ja MairionEE projekteissa oli lopulta yllättävän vähän suoraan toisiinsa verrattavaa koodia. Syitä tähän oli mm. Mairionin keskeneräisyys, erot käytettyjen kirjastojen ja ohjelmistokehysten arkkitehtuureissa, erot projektien vaatimuksissa sekä Mairionissa paremmin saavutettu ongelmien eriyttäminen (separation of concerns).

Erot projektien välillä lienee helpoin selittää käymällä läpi HTTP-pyyntöön vastaamisen prosessi.

MairionEE:ssä pyynnön käsittely on toteutettu luokan metodeina, jokaiselle polun ja HTTP-metodin yhdistelmälle oma. Esimerkiksi UserController luokan newUser metodi, hoitaa tietojen hakemisen, osan validoinnista sekä salasanan tiivistefunktion läpi ajamisen:

```
@RequestMapping(value = "/new", method = RequestMethod.POST)
public String newUser(@Validated
    @ModelAttribute("registration") RegistrationDTO
    registration, BindingResult br, HttpServletRequest request)
{
  log.info("registering a new user...");
  if (userDao.getUser(registration.getEmail()).isPresent()) {
    log.debug("e-mail address already exists: " +
        registration.getEmail());
    br.rejectValue("email", "newUser.emailExists");
  }
  if (br.hasErrors()) {
    log.debug("errors with registration: " + br.toString());
    return "newUserForm";
  } else {
    String hashedPassword =
        passwordEncoder.encode(registration.getPassword());
    User user = new User(registration.getEmail(),
        registration.getName(), hashedPassword);
    userDao.persist(user);
    HttpSession session = request.getSession();
    session.setAttribute("user", user);
    log.info("user successfully registered");
    return "redirect:/task";
  }
}
```

Mairionissa puolestaan käsittely jakautuu abstraktin ResourceService luokan metodien ja tästä perivän UserService luokan kesken. UserService määrittelee käyttäjäresurssille erityiset operaatiot, kuten HTTP-pyynnön sekä tämän sisältämän JSON-olion käsittelyn ApiRequest-olioksi ja pyynnön suorittamiseen tarvittavien MIO-operaatioiden määrittelyn. ResourceService hoitaa kaikille resursseille yhteiset toiminnot tämän ApiRequest-olion pohjalta, kuten valtuuksien tarkastamisen, virheiden

käsittelyn ja lopullisen HTTP-vastauksen rakentamisen. ResourceService luokan service metodi käsittelee pyynnön ja palauttaa vastauksen:

```
val service: AuthedService[Option[IdNumber], IO] =
AuthedService[Option[IdNumber], IO] {
  case httpRequest
  if parseHttpRequest.isDefinedAt(httpRequest) =>
    val successResponse: FallibleIO[Response[IO]] = for {
      apiRequest <- parseHttpRequest(httpRequest)
      _ <- verifyRequestIsAuthorized(apiRequest)
      results <- processRequest(apiRequest)
      response <- respond(apiRequest.opType, results)
    } yield response
    successResponse valueOrF { error => error.id match {
      case id.notAuthenticated =>
        Authentication.unauthenticatedResponse
      case id.unauthorized => Forbidden()
      case id.missingParam
         | id.unexpectedJsonMember
         | id.wrongType
         | id.notAnObject => BadRequest(error.asJson)
      case _ if debug => InternalServerError(error.asJson)
      case _ => InternalServerError()
    }}
}
```

5.2 Etujen toteutuminen

5.2.1 Tiiviys

Suoraan verrattavissa olevia koodipätkiä ei ollut käytännössä yhtään ja koodin kokonaispituutta ei ole järkevä verrata projektien vaatimuksien erilaisuuden vuoksi, joten tiiviyden vertailua ei voitu tehdä.

5.2.2 Abstraktiustaso

Kuten luvussa 5.1.3 kävi ilmi, Mairionin toteutuksessa ongelmien eriyttäminen onnistui huomattavasti paremmin. Tästä luonnollinen seuraus on, että koodi on useissa

kohdissa huomattavasti abstraktimpaa, kuten esimerkiksi todennuksen toteutus väliohjelmistona ja valtuuksien tarkistus keskitetysti ApiRequest olion pohjalta.

5.2.3 Rinnakkais-ohjelmointiin soveltuvuus

Mairionissa sivuvaikutuksia käytetään vain MIO-monadin tulkissa, muu koodi on puhtaasti funktionaalista. Puhtaat funktiot ja muuttumattomat arvot tekevät koodin rinnakkaisajosta huomattavasti helpompaa, koska puhtaat funktiot vaikuttavat toisten puhtaiden funktioiden ajamiseen vain sillä kuinka paljon prosessointiaikaa ne vievät.

MairionEE:ssä käytetään sekä muuttujia että muuttuvia tietorakenteita, mutta nämä ovat enimmäkseen rajoittuneet paikallisiin muuttujiin, mikä ei vaikuta rinnakkaisajoon. Ainoa poikkeus tästä on, että MairionEE tallentaa muuttuvan User-olion sessioon, joka on kaikkien samaan sessioon kuuluvien pyyntöjen keskenään jakama. Sivuvaikutukset rajoittuvat lokimerkintöjen tekemiseen ja JPA:n käyttöön, mutta lokimerkintöjen tekeminen ei vaikuta rinnakkais-ajettavuuteen.

Molemmissa projekteissa suurin ongelma rinnakkais-ajon kannalta on tietovaraston kanssa kommunikointi ja rajoittavana tekijänä toimii ohjelmistokehys, joka käsittelee HTTP-pyynnöt.

6 Pohdinta

6.1 Etujen arviointi

6.1.1 Yleensä

Näin jälkikäteen tuntuu varsin ilmeiseltä että kahden täysin eri teknologioilla toteutetun projektin vertailu olisi ongelmallista, vaikka ne olisivatkin tarkoitettu samankaltaisen ongelman ratkaisemiseen.

Toisaalta, imperatiivinen ja funktionaalinen ohjelmointi eroavat toisistaan niin paljon että samojen teknologioiden käyttö kummassakin projektissa pakottaisi jommankumman projektin käyttämään sen ohjelmointiparadigmalle epätyypillisiä tekniikoita, mikä puolestaan asettaisi täten saadut tulokset kyseenalaisiksi.

Vertailukategoriat olisi myös ollut aiheellista määritellä tarkemmin ja harkita etukäteen miten niitä tarkalleen ottaen verrattaisiin.

6.1.2 Tiiviys

Oma kokemukseni koodin tiiviydestä on, että funktionaaliset ohjelmointikielet ovat huomattavasti tiiviimpiä, mutta tähän vaikuttaa se, että eniten käyttämäni imperatiivinen ohjelmointikieli on Java, mikä on jopa muihin imperatiivisiin kieliin verrattuna varsin monisanainen.

6.1.3 Abstraktiotaso

Vaikka Mairion on MairionEE:hen verrattuna omasta mielestäni selkeästi abstraktimmin ohjelmoitu, on kuitenkin vaikea vetää suoraa syyseuraussuhdetta funktionaalisen ohjelmoinnin ja korkeamman abstraktiotason välille. Muita mahdollisia syitä tälle ovat:

- käytettyjen ohjelmistokehysten erot
- MairionEE:n erilaiset vaatimukset, erityisesti:
    - yksinkertaisempi valtuutus
    - ei monen henkilön työtiloja
- vuosi enemmän kokemusta ja tietoa kuin MairionEE:tä toteuttaessa
- halu tuottaa korkeampitasoista koodia opinnäytetyöhön

Omasta mielestäni korkeampi abstraktiotaso ei myöskään ole puhdas etu, vaan kompromissi. Vaikka korkealla abstraktiotasolla voi ilmaista asioita ytimekkäästi, konkreettisia asioita käsittelevää koodia on yleensä helpompi kirjoittaa ja ymmärtää.

6.1.4 Rinnakkais-ajoon soveltuvuus

Vaikka tuloksissa ei ollut paljon eroa rinnakkais-ajettavuuden suhteen, sanoisin silti että funktionaalinen ohjelmointi on tämän kannalta parempi valinta. Funktionaalinen ohjelmointi suosii sivuvaikutusten eristämistä, mikä tekee koodista rinnakkais-ajoon soveliaampaa.

Imperatiivisilla ohjelmointikielillä on tietenkin mahdollista käyttää samanlaisia tekniikoita, mutta usein sekä ohjelmointikielen standardikirjasto että kolmannen osapuo-

len kirjastot ovat suunniteltu tavalla mikä tekee näiden tekniikoiden käytön hankalammaksi.

6.2 Johtopäätökset

Tuloksista ei voi nähdäkseni tehdä mitään muuta johtopäätöstä kuin sen, että ohjelmointiparadigman etujen havaitseminen yksittäisessä projektissa on hyvin hankalaa.

Vaikka vertailukohteena oli aiemmin toteutettu samankaltainen sovellus, näiden välillä oli liikaa eroja, että voisi minkäänlaisella varmuudella sanoa tulosten johtuvan käytetyistä ohjelmointiparadigmoista.

6.3 Teknologiavalinnat

Teknologioiden valinnassa jäi yksi tärkeä kriteeri harkitsematta: verrattavuus MairionEE:ssä käytettyihin teknologioihin. Vertailun hankaluus johtui osittain tämän huomiotta jättämisestä.

Itse teknologiavalintoihin olen varsin tyytyväinen. Scalan käyttäminen on ollut positiivinen kokemus ja http4s:n kanssa ei ole ollut isompia ongelmia. Doobieta en ehtinyt alkaa käyttää, joten siitä ei voi sanoa mitään.

6.4 Funktionaalinen ohjelmointi

Tässä vaiheessa arvioisin että funktionaalisen ohjelmoinnin suurin hyöty itselleni on kasvanut luottamus omaan koodiin. Imperatiivista koodia kirjoittaessa tuntuu monesti siltä, että pientenkin muutosten tekeminen yhdessä paikassa saattaa aiheuttaa ongelmia jossain toisessa osassa sovellusta.

Funktionaalisen ohjelmoinnin käyttö on myös opettanut tunnistamaan mahdollisia ongelmakohtia rinnakkais-ajettavuuden kannalta.

6.5 Mairion

Opinnäytetyön tekemisen loppuvaiheessa löysin kolmannen tavan toteuttaa monadi, johon viitataan termillä tagless final. Tämä on kompromissi normaalin monadin ja

vapaan monadin väliltä. Se tukee useita tulkkeja, mutta sen kuvaamien toimenpiteiden tarkastelu on vapaata monadia hankalampaa, koska toimenpiteitä ei esitetä abstraktina tietorakenteena. Se myös vaikuttaa siltä, että se olisi huomattavasti helpompi ymmärtää olio-ohjelmointia aiemmin käyttäneille.

Valtuutusjärjestelmää toteuttaessa on tullut mietittyä onko se ehkä liian monimutkainen ollakseen käyttäjäystävällinen. Lupia tarvitaan normaaliin käyttöön helposti monta kymmentä. Järjestelmä itsessään tuntuu kuitenkin toimivalta, joten ehkä paras ratkaisu olisi kehittää vaihtoehtoinen yksinkertaistettu järjestelmä, joka toteutettaisiin jo kehitetyn järjestelmän puitteissa.

Aikomukseni on ehdottomasti jatkaa Mairionin kehitystä eteenpäin, keskittyen ensiksi perustoiminnallisuuksien toteuttamiseen ja sitten käyttöliittymän kehitykseen. Jossain vaiheessa täytyy myös harkita uudelleen kaikki valinnat jotka tehtiin ajan puutteen vuoksi. Vapaan monadin vaihto tagless final monadiksi on myös harkinnan alla.

6.6 Yhteenveto

Vaikka opinnäytetyön tulokset jäivät varsin puutteellisiksi, ei sen tekeminen kuitenkaan ollut turhaa. Opin paljon funktionaalisesta ohjelmoinnista, REST-rajapinnoista ja tutkimuksen tekemisestä.

Lähteet

After Decades of Neglect, Functional Programming is Finally Going Mainstream. Why Now? 2016. Artikkeli Salsita Softwaren blogissa. Viitattu 16.4.2018. http://blog.salsitasoft.com/why-now/

Alexander, A. 2017. Benefits of Functional Programming. Viitattu 14.5.2018. https://alvinalexander.com/scala/fp-book/benefits-of-functional-programming

Chiusano, P. & Bjarnason, R. 2014. Functional Programming in Scala. Shelter Island: Manning Publications.

De Goes, J. 2015. A Modern Architecture for FP. Viitattu 19.4.2018. http://degoes.net/articles/modern-fp

Hudak, P. 1989. Conception, Evolution, and Application of Functional Programming Languages. http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf

Leonard, J. 2017. The stealthy rise of functional programming. Artikkeli Computing-lehden sivustolla. Viitattu 17.4.2018. https://www.computing.co.uk/ctg/analysis/3003123/the-slow-but-steady-rise-of-functional-programming

Nanz, S & Furia, C. 2015. A Comparative Study of Programming Languages in Rosetta Code. Viitattu 23.4.2018. https://arxiv.org/pdf/1409.0252.pdf

Norris, R. 2015. SBTB 2015: Rob Norris, Programs as Values: JDBC Programming with Doobie. Luento Scala By The Bay 2015 konferenssista. Viitattu 19.4.2018. https://www.youtube.com/watch?v=M5MF6M7FHPo

Ray, B., Posnett, D., Devanbu, P. & Filkov, V. 2014. A Large-Scale Study of Programming LKanguages and Code Quality in Github. Viitattu 23.4.2018. https://cacm.acm.org/magazines/2017/10/221326-a-large-scale-study-of-programming-languages-and-code-quality-in-github/fulltext