SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

CLOJURE FULLSTACK IN A NEW PRODUCT DEVELOPMENT PROCESS

TEKIJÄ/T: Tuomas Koivistoinen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Tietotekniikan koulutusohjelma	
Työn tekijä(t) Tuomas Koivistoinen	
Työn nimi Clojure fullstack in a new product development process	
Päiväys 12.6.2020	Sivumäärä/Liitteet 39
Ohjaaja(t) Keijo Kuosmanen, Jussi Koistinen	
Toimeksiantaja/Yhteistyökumppani(t) Digiteknologian TKI-ympäristö (Digikeskus) –hanke	

Tiivistelmä

Clojure on voimakas Java Virtual Machine:lla isännöity, dynaaminen ja funktionaalinen Lisp. Clojuren pohjalta on tehty myös Javascriptiksi käännettävä ClojureScript. Kielen ympärille on rakentunut innovaatiorikas ekosysteemi teknologiaa ja ystävällinen yhteisö kehittäjiä.

Työn tarkoituksena oli esitellä, minkälaisia erikoisominaisuuksia Clojuren kaltaisella lisp:llä on selaimella ja palvelimella. Lisäksi onli tarkoitus esitellä, että minkälaista on kehittää sovelluksen käyttöliittymää, rajapintaa ja tietokantaa tietyillä Clojure ekosysteemistä löytyvillä vaihtoehdoilla.

Työssä esiteltiin Clojure ja ClojureScript ohjelmointikielet, Datomic tietokanta, EQL rajapintakieli, Pathom rajapintakirjasto, Fulcro käyttöliittymäkirjasto, sekä Shadow-cljs ja Google Closure Compiler, jotka ovat työkaluja NPM ekosysteemin hyödyntämiseen ja ClojureScriptin kääntämiseksi optimoituun JavaScriptiin. Esittelyn jälkeen on sovelluksen suunnitteluosuus, tekniikkademontraatio ja yhteenveto.

Avainsanat: Clojure, ClojureScript, Fullstack, Datomic, EQL, Fulcro, Pathom, Shadow-cljs, Google Closure Compiler, NPD

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Tuomas Koivistoinen			
Title of Thesis Clojure Fullstack in a New Product Development Process			
Date	12 June 2020	Pages/Appendices	39
Supervisor(s) Keijo Kuosmanen, Jussi Koistinen			
Client Organisation /Partners Digital Technology RDI Environment (Digi Center)			

Abstract

The goal of this thesis was to identify and demonstrate the special characteristics that a lisp such as Clojure has in the browser and on the server-side. The goal was also to demonstrate what it was like to develop an application's user interface, application programming interface and database with some of the options available in the Clojure ecosystem.

Clojure and ClojureScript programming languages, Datomic database, EDN query language, Pathom library for EQL API development, Fulcro library for UI development, Shadow-cljs for utilizing the NPM ecosystem and Google Closure Compiler for compiling ClojureScript to optimized JavaScript were studied and used.

As a result of this thesis, it was concluded that Clojure is a powerful, dynamic and functional Lisp that is hosted on the Java Virtual Machine. Clojurescript is a compile to Javascript language that is based on Clojure. The language has spawned an innovation rich ecosystem in technology and a friendly community of developers.

Keywords: Clojure, ClojureScript, Fullstack, Datomic, EQL, Fulcro, Pathom, Shadow-cljs, Google Closure Compiler, NPD

ESIPUHE

Tämä työ ei olisi onnistunut ilman mahtavaa yhteistyötä tilaajan kanssa tai apua läheisiltä, ohjaajalta ja muilta teknologioiden käyttäjiltä.

Kiitos.

Kuopiossa 25.11.2020
Tuomas Koivistoinen

SISÄLTÖ

1 JOHDANTO

Vuoden 2018 keväällä aloitin ensimmäinen työni ohjelmistokehittäjänä. Jokaisen työtehtävän jälkeen pysähdyin miettimään miten sen olisi voinut tehdä paremmin ja miten koko työtehtävältä olisi voinut välttyä. Nopeasti ajauduin tutustumaan testivetoiseen kehitykseen, olio-ohjelmoinnin suunnittelumalleihin, PHP-viitekehyksiin, SOLID-periaatteisiin, serverless-arkkitehtuuriin, cloud native-komputaatioon, CoffeeScript:iin ja Elm:iin. Muutamassa kuukaudessa koin olevani yhtä tuottava kuin moni muu kollegani nykyteknologioilla ja samalla olin ehtinyt perehtymään moneen muuhun asiaan ja tekemään kokeiluja.

Uusien teknologoiden ja periaatteiden opiskelu ja kokeilu eivät suoranaisesti kuuluneet työtehtäviini, mutta ne merkittävästi auttoivat minua suorittamaan työtehtäväni tehokaammin ja laadukkaammin, sekä auttamaan muita enemmän.

Etsin loputtomasti uusia tapoja tehdä asioita. Hylkäsin edellisiä asioita hyvin nopeasti uusien ja paremmalta vaikuttavien edessä. Vuoden 2018 kesänä tapasin funktionaalisen ohjelmoinnin, Clojuren ja Datomicin. Niitä en ole hylännyt, enkä usko koskaan hylkääväni.

Katsoin vuoden 2018 syksynä David Nolenin Om Next esityksen. Se antoi paljon ratkaisuja kokemiini ongelmiin käyttöliittymien ja taustajärjestelmien järkeilykyvyn säilyttämisen suhteen, mutta aiheutti vielä enemmän hämmennystä, koska niin moni konsepti oli täysin uusi minulle.

Kokeilin kirjastoa, mutta tutoriaalien ja dokumentaation läpikäymisenkin jälkeen en ymmärtänyt tarpeeksi, että olisin voinut olla tuottavampi sen kanssa, kuin tutumpien teknologioiden kanssa. Aikaa kului, sain lisää kokemusta React:in ja GraphQL:n kaltaisista teknologioista ja palasin esitykseen. Vihdoin aloin saamaan käsityksen kaikista palasista. Tarkastelin vaihtoehtoja ja törmäsin Tony Kayn Fulcroon ja Wilker Lucion Pathomiin.

Fulro ja Pathom jakavat samat ideat kuin Om Next, mutta Fulcro ja Pathom tekevät paljon asioita valmiiksi ja niillä on helpompi aloittaa. Fulcro ja Pathom olivat molemmat alkujaan kirjastoja, jotka syntyivät Om Nextin käytössä syntyneistä malleista. Lähdin tekemään tutoriaaleja ja vihdoin ymmärsin suurimman osan.

Olin vakuuttunut uusien konseptien paremmasta soveltuvuudesta omiin käyttötarkoituksiini ja jäin odottamaan paikkaa, missä pääsisin kokeilemaan niitä tuotantokäytössä. Aikaa kului, ja vasta 2020 keväällä löytyi hyvä kohde. Työ oli tarkoitus tehdä opinnäytetyönä ja siksi tämä raporttikin syntyi.

Olen paljon kokeneempi kehittäjä JavaScript, React, GraphQL ja SQL kaltaisilla teknologioilla, mutta nyt kun kerran olen oppinut Clojuren, Fulcron, Pathomin ja Datomicin niin palaaminen takaisin tuntuu todella kankealta ja vaikealta. Olen tehokas ja tuottava tutuilla teknologioilla, mutta suhteessa niihin, nämä uudemmat mahdollistavat aivan eri luokan potentiaalin uusien tuotteiden kehitykseen.

Funktionaalinen ohjelmointi ja Clojure ovat mullistanut käsitykseni ohjelmistokehityksestä. Datomic on mullistanut käsitykseni tietokannoista. Fulcro ja Pathom ovat mullistaneet käsitykseni käyttöliittymien ja rajapintojen kehityksestä. Opinnäytetyöni tarkoitus on kertoa näistä yleisellä tasolla, jakaa kokemuksia ja herättää lukijassa mielenkiinto oman maailman mullistukseen.

2 TEKNOLOGIAT

Sovelluksen toteutuksessa on Clojurea kaikissa kerroksissa. Jokaiseen kerrokseen on valittu myös kirjastot ja teknologiat, jotka ovat syntyneet Clojure ekosysteemissä.

2.1 Clojure

Clojure on yleiskäyttöinen ja käytännönläheinen funktionaalinen ohjelmointikieli, joka on sopiva ammattilaiskäyttöön siellä, missä sen isäntäkieli (esim. Java, C# tai JavaScript) olisi. Sen suunnitteli Rich Hickey vuonna 2005 ja se julkaistiin vuonna 2007. Clojuren tavoitteena on olla yhtä hyväksytty kieli kuin Java tai C#, mutta tukea paljon yksinkertaisempaa ohjelmointimallia esim. Rich Hickeyn kohtaamien informaatiojärjestelmien kohdalla (Hickey 2020).

Sanalla Clojure viitataan joko kieleen, joka on implementaatiosta riippumatta sama tai Java Virtual Machine (JVM) implementaatioon. Implementaatioita on kolme. Clojure käännetään JVM bytekoodiksi. ClojureScript käännetään Google Closure yhteensopivaksi JavaScriptiksi. ClojureCLR isännöidään Common Language Runtimella (CLR), joka on Microsoft .NET -viitekehityksen suoritusmoottori (Hickey 2020).

2.1.1 Isännöity

Clojure on tarkoituksella isännöity. Nuoren kielen adoptoinnissa yksi isoimmista ongelmista on kehittymätön ekosysteemi. Kehittymättömässä ekosysteemissä ei välttämättä ole juuri niitä geneerisiä ohjelmointitehtäviä varten kirjoitettuja kirjastoja, joita oman toimialakohtaisen ongelman ratkaisemiseksi tarvitsee. Logitus-, parsimis-, testaus-, http-, rajapinta- ja Excel-kirjastot ovat esimerkkejä kehittyneemmästä ekosysteemistä löytyvistä resursseista. Ekosysteemin taso on iso tekijä kielen valinnassa (Hickey 2020).

Clojuresta voi kutsua Javaa suoraan. Clojure-ohjelmassa voi siis hyödyntää sekä Javalla kirjoitettuja kirjastoja, että Clojurella kirjoitettuja kirjastoja, jotka ovat suunniteltu Java Virtual Machinelle (JVM) ja mahdollisesti itse käyttävät Javaa. Lisäksi voi hyödyntää Clojurella kirjoitettuja kirjastoja, jotka ovat suunniteltu toimimaan sekä JVM:lla, että Microsoftin Common Language Runtimella (CLR) ja lisäksi JavaScript enginellä (selain tai Node.js). Clojure tukee ”lukuehtoja” (reader conditionals), joiden avulla pääosin Clojurella toimivaan funktioon voi lisätä osia ehdollisesti alustan mukaan (Hickey 2020).

Isännöitynä Clojure pääsee myös jakamaan isäntäkielen kanssa samat optimisaatiot. Lisäksi esim. Javalle tehdyt software development kitit (SDK) ovat myös Clojuren kautta heti käytettävissä ja GraalVM kautta voi saada natiivibinäärien kaltaisen suorituskyvyn ahead-of-time kääntämisen ja edistyneiden optimisaatioiden avulla osalle Clojurea (Oracle, 2020).

Samoja etuja on myös muissa implementaatioissa. ClojureCLR avulla voi tehdä Unity pelejä ja ClojureScript avulla voi tehdä React Native mobiiliapplikaatioita, eikä se vaadi itse implementaatiolta erillisiä toimenpiteitä.

2.1.2 Informaatiomalli

Clojure on suunniteltu erityisesti informaatiosysteemien, kuten liiketoimintajärjestelmien, mallintamiseen. Informaatiosysteemin on tarkoitus pystyä keräämään tietoa, poimimaan siitä oleellisen, muuttamaan sitä toiseen muotoon, ylläpitämään sitä muutosten yli, analysoimaan sitä uuden tiedon luomiseksi, siirtämään sitä toisten käyttöön ja näyttämään se kiinnostuneille. Tämänlaisten järjestelmien informaatio on usein harvaa ja epävarmaa, inkrementaalisesti kerääntyvää ja laajentuvaa, sekä se voi olla ehdollisesti saatavilla, olla yhdistetty muun informaation kanssa ja esiintyä mielivaltaisissa konteksteissa (Hickey 2020).

Informaatiosysteemien mallintamista voi verrata keinotekoisten systeemiem, kuten kielten kääntäjien, mallintamiseen. Keinotekoiset systeemit eroavat tästä siinä, että niissä voidaan luoda omat säännöt informaation käsittelylle, poistaa siitä epävarmuudet, kieltää kaikki epäsäännöllisyydet, sekä määritellä etukäteen, missä konteksteissa informaatiota voi esiintyä. Keinotekoisten systeemien harvoin tarvitsee jakaa omaa tietoansa muulle maailmalle. Jos tietoa tarvitsee jakaa, niin haastenaa voi olla esimerkiksi etukäteen päätetyn informaation kontekstisidonnaisen semantiikan jakaminen informaation kuluttajalle (Hickey 2020).

Staattisesti tyypitetyt kielet, joiden päärakenne tiedon mallintamiseen on luokka tai tallenne, eivät ole ainoa tapa mallintaa informaatiosysteemejä. Tässä tavassa informaation aggregaatio itsessään on isoin ajuri informaation semantiikan suhteen, sen sijaan että aggregaatiota pidettäisiin vain kontekstisidonnaisina informaation esiintymänä. Lisäksi esiintyvä informaatio on usein täysin etukäteen lueteltu, ne ovat suljettu laajennukselle, niitä on hankalampi yhdistää muun informaation kanssa ja kerätä inkrementaalisesti, informaatio on nimetty vain kyseiseen systeemiin sopivaksi sekä tämä lähetysmistapa voi vaatia erillisen työkalun informaation mallin ja datan suhteiden kartoitukseen, kuten object-relation mapping (ORM) työkalun (Hickey 2020). ORM-työkalun käytössä voi tulla ongelmaksi object-relational impedance mismatch, koska olio-ohjelmoinnin tyyli mallintaa dataa eroaa niin paljon esimerkiksi relaatiotietokantojen tyylistä mallintaa dataa.

Clojuren lähestymistapa on erilainen. Informaation semantiikka annetaan attribuuttitasolle aggregaattitason sijaan. Attribuutit pyritään nimeämään tavalla, jolla ne ovat suoraan käytettävissä muissakin järjestelmissä, esimerkiksi com.youtube.user/name. Informaatiota voidaan aggregoida kontekstisidonnaisesti, mutta silti säilyttää attribuuttatason semantiikka. Aggregaatiot itsessään ovat usein vain assosiatiivisia tai peräkkäisiä datarakenteita, joihin pystytään keräämään informaatiota, poimimaan siitä olennainen, muuttamaan sitä toiseen muotoon, ylläpitämään sitä muutosten yli, analysoimaan sitä uuden tiedon luomiseksi, siirtämään sitä toisten käyttöön ja näyttämään se kiinnostuneille. Informaatiota on tällä tavalla helppo kerätä inkrementaalisesti, koostaa

kontekstisidonnaisesti, yhdistää muun informaation kanssa, käsitellä vain ehdollisesti saatavilta osin ja laajentaa kun opitaan uutta (Hickey 2020).

2.1.3 Funktionaalinen

Funktionaalisen ohjelmoinnin voi asetella vastakkkain imperatiivisen ohjemoinnin kanssa. Imperatiivisessa ohjelmoinnissa on tyypillistä kuvailla valtaosa ohjelman logiikasta tilaa muuttavien lausuntojen avulla. Vastakohtaisesti funktionaalisissa kielissä on tyypilllistä kuvailla valtaosa ohjelman logiikasta arvoja palauttavien lausekkeiden avulla.

Lähestymistavat ovat pienessä skaalassa hyvin samankaltaiset. Merkittävä ero syntyy skaalan kasvaessa, ja jos imperatiiviset lausunnot eivät ole vierekkäin, vaan kaukana toisistaan. Lausunnot voivat tapahtua eri luokkien eri metodeissa ja niiden välillä voi tapahtua tuhansia muitakin asioita, joista jokainen voi olla vastaukseen merkittävä asia. Imperatiivinen lähestymistapa sitoo yhteen arvon ja ajan luomalla tilan, joka voi olla merkittävä kompleksisuuden lähde monissä järjestelmissä (Moseley ja Marks 2006).

Kun ajanhetkiä ja paikkoja on tuhansia, niin lausekkeiden käytön hyödyt alkavat näkymään. Siinä missä lausunnoilla on helppo sijoittaa arvoja jaettuihin paikkoihin toisten arvojen päälle eri ajanhetkinä, ja näin sitoa yhteen aika, paikka ja arvo, niin lausekkeet tarvitsevat toimiakseen arvoja ja palauttavat aina uusia arvoja. Valtaosa funktionaalisen ohjelman lausekkeista on puhtaita ja deterministiä. Ne siis vain ja ainoastaan kuvaavat arvojen välisiä suhteita ajasta ja paikasta riippumatta, eivätkä tee itse sivuvaikutuksia. Lausekkeista koostetaan funktioita. Valtaosa funktioistakin pyritään rakentamaan puhtaiksi ja deterministisiksi. Kun herää kysymys, että mitä tämän rivin puhdas funktio palauttaa arvoksi, niin sitä voi kutsua minä tahansa ajanhetkenä samoilla arvoilla ja saada saman palautusarvon. Puhtaista funktioista pyritään rakentamaan lisää puhtaita funktioita. Funktiot kuvailevat arvojen välisiä suhteita, mutta ovat myös itse arvoja. Funktiot voivat siis olla argumentteina toisiin funktioihin tai olla toisen funktion palautusarvona. Funktioita voi koostaa ja käyttää, kunnes kaikki tarpeelliset arvojen väliset suhteet on kuvailtu ja palautettujen arvojen perusteella voidaan toteuttaa tilanteenmukaiset sivuvaikutukset.

Pienessä mittakaavassa tila on usein vielä hallitavissa, mutta mittakaavan kasvun myötä on hyvin nopeasti mahdoton järkeillä ohjelman tuottamia vaikutuksia. Järkeilyn mahdollisuuden menettäminen vaikeuttaa muutoksien tekemistä ja pakottaa jokaisen käyttötapauksen raskaaseen testaamiseen. Testaus ei ole determinististä, joten vikatilanteiden toisto on hankalaa ja läpäissyt testi ei takaa koodin toimivuutta toisena ajanhetkenä. Muutoksien tekeminen voi olla vaikeaa, koska on usein vaikea tietää, rikkooko yhden paikan muutos jonkin toisen paikan.

Funktionaalisella tyylillä tehdyissä järjestelmissä tilan käyttö on hyvin kurinalaista ja sivuvaikutuksien suhteen ollaan hyvin varovaisia. Puhtaat funktiot vain kuvaavat arvojen välisiä suhteita. Niiden toimivuutta on usein helpompi järkeillä, koska ei tarvitse ottaa huomioon ulkoisten riippuvuuksia tilaa,

vaan tarvitsee ottaa huomioon vain funktiolle annetut arvot. Testaaminen on myös todella kevyttä, koska eri tilanteita voidaan simuloida käyttämällä eri arvoja, sen sijaan että jouduttaisiin luomaan eri tiloja tai keinotekoisia toimintoja.

2.2 Lisp

Clojure kuuluu Lisp kielten perheeseen ja sellaisena erottuu huomattavasti muista ohjelmointikielistä symbolisten lausekkeiden (S-expression) ja niiden etuliite notaation (prefix notation) vuoksi, joita käytetään sekä Clojuren dataformaatissa (extensible data notation / edn), että itse lähdekoodissa. Clojure on homoikoninen, eli yksi Clojure ohjelma voi lukea toisen Clojure ohjelman ja muokata sitä kuten muokkaisi mitä tahansa dataa (Hickey 2020).

2.2.1 Syntaksi

Kokonaisluku määritetty tarkkuus (64-bit)	1234
Kokonaisluku mielivaltainen tarkkuus	123456789N
Double	1.333
BigDecimal	1.333M
Osuus	4/3
Merkkijono	”tuomas”
Merkki	\t \u \o \m \a \s
Symbooli	tuomas, esimerkki.nimitila/tuomas
Avainsana	:tuomas, :esimerkki.nimitila/tuomas
Boolean	true, false
Ei mitään	nil
Regex-kaava	#”\d+”
Linkitetty lista (list)	(1 2 3), (eka toka kolmas), (+ 1 2 3)
Indeksoitu vektori (vector)	[1 2 3], [eka toka kolmas], [+ 1 2 3]
Assosiatiivinen kokoelma (map)	{:eka 1 :toka 2 :kolmas 3} {1 ”eka” 2 ”toka”}
Distinktien arvojen kokoelma (set)	#{:eka :toka :kolmas}
Merkattu literaali (tagged literal)	#inst ”1993-17-09T23:20:50.52Z” #uuid ”87dd805b-8f55-4f2e-bf4e-d10f92c28871”

Clojure ohjelman lähdekoodi on Clojure lista, jonka sisällä muita listoja. Listoissa on dataa, joka on eroteltu välilyönneillä ja mahdollisesti myös pilkuilla (pilkut tulkitaan välilyönteinä). Lista on lauseke, jonka ensimmäinen jäsen on kutsuttava (invokable) ja loput jäsenet ovat argumentteja kutsuttavalle.

```
(+ 1 (* 2 2))
; => 5
```

Kutsuttavat jäsenet voivat olla funktioita, mutta myös muita kutsuttavuus abstraktion implementoivia datatyyppejä, kuten assosiatiivisia kokoelmia (map) tai avainsanoja (keyword). Esimerkiksi avainsanan

implementaatio ottaa argumenteikseen pakollisena assosiatiivisen kokoelman, josta se etsii itseään, ja valinnaisen oletusarvon toiseksi argumentiksi (Hickey 2020).

```
(:sukunimi {:etunimi ”Tuomas”} ”Ei löydy”)
; => ”Ei löydy”
```

2.2.2 Makrot

Lukija lukee Clojure lähdekoodin merkkeinä ja tekee siitä datarakenteita kääntäjälle. Kääntäjä ottaa datarakenteet ja tekee niistä Java bytekoodia. Clojuressa on pieni määrä erikoismuodostelmia, jotka kääntäjä ymmärtää suoraan, mutta suurin osa Clojuresta on kirjastokoodia, joka on kirjoitettu erikoismuodostelmien ja muun kirjastokoodin avulla (Hickey 2020).

Clojuressa on makrosysteemi, jonka avulla toi tehdä syntaktisia abstraktioita. Makrot ovat kuin funktioita, paitsi että ne ottavat argumenttinsa datana ilman evaluointia ja päättävät itse evaluoinneista. Niiden avulla kieltä voi laajentaa, ilman että kielen ytimeen tekee muutoksia. Tämä on yksi syy siihen, että Clojuren ydin pysyy pienenä ja suurin osa kehityksestä tapahtuu kirjastoissa (Hickey 2020). Esimerkki mukavamman syntaksin antavasta makrosta voisi olla -> makro, joka evaluoi muodostelmansa (forms) yksi kerrallaan ja siirtää edellisen palautusarvon seuraavan ensimmäiseksi argumentiksi. Useimmissa kielissä tämä vaatisi muutoksia itse kieleen.

```
(- (* (+ (/ 5 4) 3) 2) 1)
```

```
(-> 5 ; aloita viidellä ja sitten
  (/ 4) ; jaa neljällä
  (+ 3) ; lisää kolme
  (* 2) ; kerro kahdella
  (- 1)) ; vähennä yksi
```

2.2.3 REPL

Clojurella työskennellessä on tavanomaista käyttää REPL:ia (Read Eval Print Loop). REPL on kuin interaktiivinen komentokehoite, mutta eroaa siitä huomattavasti ja on reilusti interaktiivisempi. Useimmissa ohjelmointikielissä koodi kirjoitetaan tiedostoihin, ohjelma käynnistetään, tiedostoja muokataan ja ohjelma käynnistetään uudelleen. Clojuren tapauksessa koodi kirjoitetaan tiedostoihin, ohjelma käynnistetään, mutta lisäksi käynnistetään REPL sessio, joka on yhdistetty suoraan suoritusajan ympäristöön. Kirjoitettua koodia evaluoidaan muodostelma kerrallaan, usein jollain näppäinyhdistelmällä, ja sen seurauksena REPL ikkunaan tulostetaan evaluoinnin tulos. Evaluoinnin yhteydessä voi olla myös sivuvaikutuksia, jotka tapahtuvat suoristusajan ympäristössä.

Ohjelma siis tavanomaisesti pidetään käynnissä koko ajan. Ohjelman suoritusajan ympäristössä suoritetaan lisää koodia muodostelma kerrallaan. Koodia suorittamalla pystyy suorittamaan rajapintakyselyitä, tietokantakyselyitä, funktioita ja vaikka käynnistämään http-palvelimen. Kehittäjä ei siis joudu suorittamaan koko ohjelmaa ja yrittämään selvittää yksittäisten kohtien toimintaa taukopisteillä tai lokittamalla, vaan hän voi yksinkertaisesti suorittaa haluamansa kohdat haluamillaan arvoilla suoraan editorista.

2.3 ClojureScript

ClojureScript on Clojuren implementaatio, joka käännetään Google Clojure kääntäjän edistyneelle koko-ohjelma optimisaatiolle yhteensopivaksi JavaScriptiksi. ClojureScript mahdollistaa Clojuren käytön siellä missä Javaa ei voi käyttää ja pääsyn uuteen ekosysteemiin. ClojureScriptiä kirjoitetaan selaimelle, Node.js:lle ja React Nativelle. Verrattuna JavaScriptiin, ClojureScript tarjoaa pysyvät ja muuttumattomat datarakenteet, automaattisen Google Closure kääntäjän edistyneet koko-ohjelma optimisaatiot, ”koodi on dataa” filosofian ja makrot, sekä suurimman osan Clojuren ominaisuuksista. ClojureScript eroaa Clojuresta esimerkiksi siten, että monisäie (multithread) ohjelmointia ei voi tehdä ja Java tietotyyppien sijaan käytetään JavaScriptin tietotyyppejä. JavaScript moottorit ovat yhtä lailla Java Virtuaalikoneiden kanssa kohde laajalle tutkimustyölle ja optimisaatioille, eli suorituskyky on kilpailukykyinen.

2.3.1 NPM

NPM on paketinhallintatyökalu JavaScript maailmassa ja maailman isoin ohjelmistorekisteri. ClojureScriptilla kehittäessä on tavanomaista hyödyntää NPM-paketteja.

2.3.2 Google Closure kääntäjä

Google Closure kääntäjä tekee JavaScriptistä nopeampaa ladata ja suorittaa (Closure Compiler 2020). Edistynyt koko-ohjelma optimointi suorittaa lukuisia optimisaatioita lukuisia kertoja, koska yksi optimisaatio voi mahdollistaa toisia optimisaatioita. Optimisaatioihin kuuluu esim. käyttämättömän koodin poistaminen, uudelleen nimeäminen, funktioiden riville sijoittaminen ja ylimääräisyyksien poistaminen.

```
var FunctionForIntegersOnly = function(int1, int2){
   return int1 + int2;
}

var FunctionForStringsOnly = function(str1, str2){
   return str1 + str2;
}

alert(FunctionForIntegersOnly(1, 2) + FunctionForStringsOnly("a", "b"));


// kehittynyt optimisaatio
alert("3ab");
```

2.4 EQL ja Pathom

EQL, eli EDN Query Language on spesifikaatio, jonka voidaan tehdä deklaratiivisia, hierarkisia, sisäkkäisiä valintoja datasta. EQL on nimensä mukaisesti kyselykielimutta sillä ei ole omaa kieltään, vaan se käyttää Clojuren data formaattia EDN:nia (extensible data notation).

EQL kyselyssä on EDN muodossa kuvailtuna tarvittu data. Kyselyn vastaanottavassa palvelussa on resolvereita, joissa on määritetty niiden tarvitsema ja tuottama data. Palvelussa voi olla resolveri, joka ei tarvitse mitään dataa, ja palauttaa :user/id esimerkiksi käyttäjän autentikaatio tokenista. Lisäksi voi olla resolveri, joka tarvitsee :user/id, ja palauttaa :user/name esimerkiksi tietokannasta. Tässä tapauksessa kysely [:user/name] palauttaa käyttäjästä riippuen oikean nimen esimerkiksi {:user/name ”Tuomas”}.

Sisäkkäisyys (nesting) kuvaillaan kokoelman (map) avain arvo pareilla (join). Esimerkiksi käyttäjän nimen ja yrityksen tiedot voi saada kyselyllä [:user/name {:user/company [:company/id :company/name]}]. Kysely voi palauttaa esimerkiksi {:user/name ”Tuomas” :user/company {:company/id 1 :company/name ”Talentree”}}. Huomion arvioista tässä esimerkissä on, että :user/company voi olla yksittäinen kokoelma tai lista kokoelmia. Ero käy usein ilmi vain nimeämiskäytänteistä.

Syöttö dataa voi laittaa kyselyyn esimerkiksi identillä (ident = identifoiva attribuutti + arvo) tai parametrina. Identillä kysely koskee tiettyä entiteettiä esimerkiksi [{[:user/id 1] [:user/name]}]. Parametreja voi antaa EDN-listan avulla esimerkiksi [(:user/height {:unit :meter})]. Lisäksi EQL avulla voi tehdä esimerkiksi rekursiivisia kyselyitä, joissa on mielivaltainen määrä sisäkkäisyyttää [:fs/name {:fs/folders …}].

EQL on verrattavissa GraphQL esimerkiksi siten, että molemmilla voi kuvailla mielivaltaisen sisäkkäisiä datarakenteita, molemmat voi parametrisoida ja molemmissa voi tehdä yhdistelmä (union) kyselyitä ja haarauttaa kyselyn data jonkun tekijän perusteella. EQL eroaa esimerkiksi siten, että GraphQL:ssa on staattinen tyyppijärjestelmä, jota se vaatii toimiakseen ja EQL:ssa ei ole. Toinen iso ero on, että GraphQL kysely on merkkijono ja EQL on dataa. Esimerkiksi tämän vuoksi on GraphQL ominaisuuksia, jotka eivät ole tarpeellisia EQL:ssa, kuten GraphQL Fragmentit, joilla voi koostaa isoja kyselyjä pienistä paloista. EQL on jo itsessään dataformaatti, joka on jo koostettavissa datan palasista.

Koostavat rajapinnat, joiden avulla datan tarpeen määrittely saadaan siirrettyä datan käyttäjälle, ovat tärkeitä kehityksen tehokkuudeen ja sovellusten suorituskyvyn kannalta. Sen sijaan, että käyttäjä joutuvat itse koostamaan tarvitsemansa datan useilla kyseillä ja mahdollisesti odottamaan kyselyiden välillä, niin koostavien rajapintojen avulla saadaan siirrettyä koostaminen palvelimelle, missä se on tehokkaampaa ja missä useampi käyttäjä voi hyötyä kirjoitetusta logiikasta.

EQL vaikuttaa olevan potentiaalisin RDF (resource description framework) tyylisen lähetystymistavan vuoksi, jossa datan semantiikka on fieldi ei aggregaatti tasolla. Tämä mahdollistaa esim. triviaalin rajapintojen yhdistäminen tiettyjen attribuuttien kohdalla. Entiteetillä voi olla :fi.tuomaskoivistoinen/username, jonka avulla saadaan tuomaskoivistoinen.fi järjestelmän dataa, kuten :fi.tuomaskoivistoinen.youtube.channel/id, joka voidaan yhdistää youtube.com järjestelmän kanssa kohdasta :com.youtube.channel/id. Tämä myös vähentää resolvereiden määrää, koska GraphQL skeemassa on usein samaa dataa tarkoittava fieldi monessa tyypissä, mutta se pitää manuaalisesti aina yhdistää oikeaan resolveriin.

EQL rajapintoja voi tehdä Pathom-kirjaston avulla. Cljc-kirjastona, sillä voi tehdä rajapinnan selaimeen, NodeJS:lle tai JVM:lle. Jos rajapinta käännetään JavaScriptiksi, niin kyselyt voidaan resolvata synkronisesti tai asynkronisesta. JVM:llä resolvaus onnistuu myös monisäikeisesti.

2.5 Fulcro

Fulcro on fullstack kehitykseen tarkoitettu kirjasto. Sen avulla voidaan kirjoittaa tilaa käyttäviä (stateful) React-komponentteja deklaratiivisemmalla tyylillä, koostaa komponenttien datatarpeista automaattisesti EQL-kyselyitä, normalisoida data käyttöliittymän tietokantaan ja antaa se komponenteille niiden tarvitsemassa muodossa, sekä suorittaa mutaatioita, joissa on määritetty optimistiset ja pessimistiset toimenpiteet, sekä minkä komponentin dataa mutaatiosta palaa automaattista datan normalisaatiota varten.

2.5.1 Tilaa käyttävät komponentit

Tilaa käyttävissä Fulcro komponenteissa määritellään kokoelma (map) dataa, jossa on määritetty komponentin tarvitsema data ja mahdollisesti komponentin ident, lapsikomponenttien tarvitsema data, alkutila (inital state) ja lukuisia muita asioita. Määritellyt asiat ovat vain dataa ja dataa voi laajentaa omiin käyttötarkoituksiin, esim liitännäisten (plugin) käyttötarpeita varten. Tämän lisäksi määritellään renderöitävä ohjelmakoodi (render body), jossa määritellään mitä käyttöliittymässä pitäisi tämän komponentin kohdalla näkyä.

Komponentin määrittelyssä on siis kaksi osaa. Kokoelma avain arvo pareja, jossa määritellään komponenttiin liittyvä data ja muita ominaisuuksia, sekä itse renderöitävä ohjelmakoodi. Funktionaalisella lähetysmistavalla käyttöliittymä pyritään rakentamaan datan funktioksi ja Fulcron tilaa käyttävillä komponenteissa se onnistuu selkeästi.

2.5.2 Data tarpeiden koostaminen ja automaattinen normalisaatio

Tilaa käyttävissä komponenteissa määritellään komponentin datatarpeet. Määriteltyjä datatarpeita voidaan käyttää datan hakemiseen rajapinnan kautta. Data voidaan hakea käyttöliitymän (client) tietokantaan juuritasolle antamalla juuri tason avain esimerkiksi :orders tai hakea ja automaattisesti

normalisoida identin perusteella identifioivan attribuutin mukaan nimettyyn taulun identiteetin arvon kohdalle esimerkiksi :order-list/id tauluun 1 kohdalle.

Datan voi myös kohdistaa johonkin olemassaolevan entiteetin attribuuttiin esimerkiksi tilauslistan voi kohdistaa :user/id tauluun 1 kohdalle :user/orders attribuuttiin. Jos data on ident perusteella automaattisesti normalisoitavissa, niin se normalisoidaan ja jos samalla halutaan kohdistaa data toisen entiteetin attribuuttiin, niin silloin kohdistuskohtaan tulee vain ident viittaus esimerkiksi [:order-list/id 1].

Datatarpeita käytetään myös komponenttien uudelleen renderöintiä varten. Jos usealla komponentilla on datatarpeissa tietyn käyttäjän tietoja ja sen käyttäjän tiedot muuttuvat, niin muutos näkyy välittömästi kaikkien komponenttien kohdalla, koska komponentti renderöidään uudelleen muuttuneella datalla.

Taustajärjestelmissä on kauan aikaa tajuttu säilyttää dataa tietokannoissa, normalisoida se tauluihin ja duplikoinnin sijaan käyttää viittauksia. Käyttöliittymissä vastaava lähestymistapa ei ole aina ollut mielekästä, mutta esimerkiksi Fulcron kaltaisella kirjastolla se on käytännöllistä ja tehokasta.

2.5.3 Mutaatiot

Fulcron mutaatiot ovat tapa poistua puhtaasta ja funktionaalisesta maailmasta tekemään likaisia sivuvaikutuksia, mutta mahdollisimman puhtaasti. Mutaatiossa voidaan määritellä optimistiset toimenpiteet, ehdot mutaation lähettämiselle rajapinnalle, pessimistiset toimenpiteet ja muita asioita.

Optimistinen toimenpide on ympäristön funktio, joka palauttaa uuden ympäristön. Ympäristössä on muunmuassa sovelluksen tiedot ja käyttöliittymän tila. Optimistisessa toimenpiteessä voidaan esimerkiksi lisätä tilaan entiteettejä, tehdä muokkauksia tai poistaa dataa. Mutaatio lähetetään ehdollisesti rajapinnalle.

Ehto on ympäristön funktio, mutta voi tietenkin myös olla aina tosi tai epätosi. Ehdossa voidaan myös määritellä mikä entiteetti mutaation vastauksena tulee automaattista normaalisaatiota varten ja myös kohdistaa se jonkin toisen entiteetin attribuuttin suoraan tai viittauksena.

Pessimistinen toimenpide on ympäristön ja vastauksen funktio ja palauttaa uuden ympäristön. Ympäristöstä voidaan esimerkiksi poistaa latausmerkkejä (loading markers), jonka avulla indikoidaan vastauksen saapuminen tai tyhjentää syöttökenttiä.

Mutaatiot yksinkertaistavat käyttöliittymän kautta vaikutuksien aikaansaamista. Samaan mutaatioon voidaan vaivatta määritellä optimiset toimenpiteet, palautuvat datan sijoitus käyttöliittymän tietokantaan ja pessimistiset toimenpiteet. Vastaavat toimenpiteet ovat perinteisimmillä teknologioilla työläämpiä ja aiheuttavat mielestäni enemmän kognitiivista kuormaa muutoksien teon yhteydessä.

2.6 Shadow-cljs

Shadow-cljs on vaihtoehtoinen JavaScript pakettien rakennustyökalu ClojureScriptistä. Shadow-cljs tekee myös NPM ekosysteemin käytön ClojureScriptin sisällä paljon helpommaksi. NPM paketit asennetaan tavalliseen tapaan ja niitä voi käyttää suoraan ClojureScriptin sisällä.

ClojureScriptiä käyttäessä on yleistä käyttää tilan säilyttävää koodin uudelleenlataamista selaimeen. JavaScriptillä kehittäessä käytetään uudelleenlataamista, jonka yhteydessä selaimen sivu ladatataan uudelleen tilan nollaamiseksi ja virheiden estämiseksi. Funktionaalisella tyylillä kehittäessä käyttöliittymä on tilan funktio. Koodia muuttaessa muokataan funktioita, mutta tilaa ei tarvitse nollata. Muutokset saa siis näkyväksi välittömästi ja selaimen sivua päivittämättä. Shadow-cljs on yksi työkalu, joka mahdollistaa tilan säilyttävän koodin uudelleenlataamisen.

Shadow-cljs avulla voi tehdä myös muita asioita, mitä vastaavalta työkalulta JavaScriptissa odottaisi, kuten ympäristömuuttujien käyttö, koodin automaattinen uudelleenlataaminen selaimeen ja koodin suoritus latauksen yhteydessä.

2.7 Datomic Cloud

Datomic Cloud on hajautettu tietokanta, joka tarjoaa ACID (atomic, consistent, isolated, durable) transaktiot, joustavan skeeman, datalog kyselyt, täyden historian datasta ja tuen SQL-analytiikalle. Tietokannalla on korkea saatavuus, se skaalaa horisontaalisti, integroituu AWS parhaiten turvallisuus käytänteiden kanssa ja voi toimia koko järjestelmän infrastruktuurina Ionien avulla.

2.7.1 Datomic tietomalli

Datomicissa ei ole tietokantatauluja, vaan universaalinen skeema käyttäjän määrittelemiä attribuutteja, joita vasten voi tehdä kyselyjä. Tietokanta koostuu datomeista, joissa on määritelty EAVTO (entity, attribute, value, transaction, operation) arvot.

Entiteetti on yksi olio tietokannassa, jolla voi olla mielivaltainen määrä mitä tahansa skeemassa määriteltyjä attribuutteja. Entiteettin arvo datomissa on numeraalinen ja tietokannalle uniikki. Attribuutti on skeemaan määritelty nimetty ominaisuus tietyllä tyypillä ja kardinaliteetillä. Arvo on attribuutin tyypin mukainen arvo. Transaktio on entiteetti itsessään ja sen avulla voidaan liittää datomit tiettyyn transaktioon ja tiettyyn ajanhetkeen. Operaatio on joko lisäys tai poisto. Myös tiedon poistoa käsitellään lisättynä faktana tietokantaan.

Perinteisissä tietokannoissa on taulut, mihin lisätään rivejä, mistä poistetaan rivejä ja missä muokataan rivejä. Ei ole triviaalia pyrkiä saamaan käsitystä siitä, että mitä tietoa on tullut millonkin, onko sitä muokattu, onko mitä poistettu ja miksi. Usein näkee rivikohtaisesti luotu tai muokattu kolumnit, jotka voivat antaa suuntaa, mutta näitäkin tietoja voi muokata, ne eivät kerro kuin rivitasolla,

että mitä on luotu ja mitä on muokattu, ja rivin poiston yhteydessä nämäkin tiedot katoavat. Jollain aloilla on laissa säädetty, että aukoton kirjausketku (audit trail) on oltava saatavilla, mutta ongelmatilanteissa siitä olisi hyötyä alasta riippumatta. Pilvipalveluiden aikakautena datan säilyttäminen on niin halpaa, että sitä ei välttämättä kannata turhaan poistaa tai ylikirjoittaa.

Semanttisesti Datomic on järjestelmä, joka vain akkumuloi faktoja. Se ei kuitenkaan ole järjestelmä, jossa vain lisätään dataa ja olemassa olevaan dataan ei kosketa, eikä jaa siihen samaa suorituskyvyn luonnetta. Kaikkea dataa voi kohdella muuttumattomana, eli välimuistituksen saa kokonaan ilman tai hyvin vähällä konfiguraatiolla ilman vaikutuksia transaktionaalisiin takeisiin. Dataa jää automaattisesti välimuistiin moneen eri tasoon ja mikään välimuistin data ei vanhene koskaan, koska ne ovat vain menneisyyden faktoja, jotka ovat aina totta.

2.7.2 Datomic kyselyt

Datomic kyselyt tehdään joko datalog tai pull kyseinä. Datalog kyselyssä voi myös käyttää pull syntaksia. Datalog on deklaratiivinen logiikka ohjelmointi kieli. Siinä määritellään mitä halutaan löytää ja mitkä asiat pitää olla tosi löydettyjen asioiden kohdalla. Ehdoissa voi olla entiteetti, attribuutti, arvo, transaktio ja operaatio, eli koko datomi. Jos etsitään ?user ja annetaan ehdoksi [?user :user/name ”Tuomas”], niin vastauksessa on kaikki entiteetit, joilla attribuutin :user/name arvo on Tuomas. Jos etsitään ?lastname ja annetaan ehdoiksi [?user :user/name ”Tuomas”] [?user :user/lastname ?lastname], niin vastauksessa on samojen entiteettien :user/lastname arvot, jos niillä on kyseinen attribuutti. Jos ehdot kasvavat pitkäksi ketjuksi, niin ne voi yhdistää säännöksi esimerkkinä [(user-info ?user ?name ?last ?company)<monta vektoria ehtoja>]. Jos user-info sääntöä käyttäessä on tietty ?user, niin löytää sen ?name ?last ?company. Jos on tietty ?company, niin löytyy kaikki ?user ja niiden ?name ja ?last. Jos on tiedossa vain ?name ja ?last, niin silloin löytyy kaikki mahdolliset ?user ja ?company.

EQL on pull syntaksi. inspiroima. Pull syntaksi on tapa tehdä deklaratiivisia informaatio valintoja entiteeteistä. Sillä sovelletaan mallia (pattern) yksittäiseen entiteettiin tai kokoelmaan. Vastauksena on mallin muotoinen kokoelma dataa. Esimerkiksi jos edellisen esimerkin ?user esimerkkiin soveltaisi mallia [:user/name :user/lastname], niin vastaus voisi olla {:user/name ”Tuomas” :user/lastname ”Koivistoinen”}. Pull syntaksissa voi myös valita kaikki attribuutit tähdellä [*], kuvailla mielivaltaisesti sisäkkäisyyksiä [{:user/orders [{:order/items [:item/name]}]}], kuvailla rajallista ja rajatonta rekursiota [:user/name {:user/friends 5}] [:user/name {:user/friends …}], hakea muita entiteettejä joissa on itse viittauksena käänteisesti attribuutin nimellä [:user/name :company/_employees] ja tehdä lukuisia muita asioita.

Datomic tietokantaan tarvitaan yhteys. Yhteyden läpi saa haettua muuttumattoman tietokanta arvon tiettynä ajanhetkenä. Kyselyt tehdään muuttumatonta arvoa vastaan. Jos faktoja halutaan viedä tietokantaan, niin ne viedään transaktiona suoraan yhteyden läpi. Jos vietyä dataa vasten halutaan

tehdä kyselyjä, niin transaktion vastauksesta voi ottaa uuden tietokannan arvon tai sen voi hakea yhteyden läpi.

Transaktion data voi olla vektori kokoelmia, joihin voi valinnaisesti laitaa tilapäisen identiteetti (tempid) arvon [{:db/id ”uusi-kayttaja” :user/name ”Tuomas”}]. Tilapäistä identiteetti arvoa käytetään transaktion datan kuvailuun ja sillä saa vastauksesta tietokantaan annetun identiteetti arvon (-> result :tempids (get ”uusi-kauttaja”)). Transaktio data voi olla myös vektori datomien lisäyksiä ja poistoja [[:db/add ”uusi-kauttaja” :user/name ”Tuomas”][:db/add ”uusi-kayttaja” :user/lastname ”Koivistoinen”] [:db/add ”toinen-kayttaja” :user/name ”Jokumuu”][:db/retract ”toinen-kayttaja” :user/attribuutti]]. Jos attribuutin kardinaliteetti on yksi, niin lisäykset tekevät myös implisiittisen poiston. Poistolle voi antaa vapaaehtoisesti myös poistettavat attribuutin arvon. Transaktiossa voi antaa attribuutteja ja arvoja myös erikoisella ”datomic.tx” tilapäisidentiteetin arvolla, jolloin faktat lisätään kyseisen transaktion transaktio entiteettiin [:db/add ”datomic.tx” :db/doc ”manuaalinen virheellisten arvojen korjaus”].

Tavanomaiset kyselyt käsittelet vain tietoa mikä on kyseisen tietokannan hakemisen ajanhetken kohtana totta. Tavanomaisten kyselyjen lisäksi voi tehdä kyseilyitä toiselle ajanhetkelle (as-of query), jolloin kysely käsittelee tietoa, joka oli sinä ajanhetkenä totta. Lisäksi voi tehdä kyselyn koko tietokannan historialle, jolloin vastauksena saa kaikki lisätyt ja poistetut (retracted) tiedot, jotka täyttävät kyselyn ehdot esimerkiksi voidaan etsiä ?tx (transaktio) ?nimi ?op (operaatio) ehdoilla [?user :user/name ?nimi ?tx ?op], jolloin vastauksena saa faktoja liittyen entiteetteihin, joilla on :user/name attribuutti.

3 SUUNNITTELU

Sovelluksen suunnittelu vaiheessa asiakkaan kanssa on jäsennelty liiketoiminnallisia tavoitteita ja evaluoitu eri teknologoita. Toiseksi vaihtoehdoksi lopulta jäi AWS Amplify työkaluilla ja palveluilla tehty Serverless-arkkitehtuurinen toteutus, mutta Datomic Cloud mahdollistama tietokanta- ja sovellusinfrastruktuurin jakaminen ja funktionaalinen lähetystymistapa mahdollistivat arvioiden mukaan pienemmät kokonaiskustannukset ja nopeamman sovelluksen valmistumisajan markkinoita varten. Jos sovellus olisi kohdistettu kuluttajille ja kansainvälisille markkinoille, niin toteutus olisi tehty Serverless-arkitehtuurilla vaivattoman ja täysin elastisen skaalautumisen vuoksi.

3.1 Tavoitteet

Sovelluksen tavoitteena oli virtaviivaistaa asiakkaiden liiketoiminnan prosesseja ja loppujen lopuksi myös olla myytävä tuote muille yrityksille. Henkilökohtaisia tavoitteita minulla oli yrittäjyyden harjoittelu ja löytää uusia tapoja toteuttaa yhä laadukkaampia ominaisuuksia ja yhä tehokkaammin.

3.2 Vaatimukset

Sovelluksen pitää pystyä virtaviivaistamaan yksittäinen asiakkaan liiketoimintaprosessi. Lisäksi pitää olla mahdollisuus hallita oman organisaation käyttäjiä ja luomaan uusia käyttäjäorganisaatioita suoraan käyttöliittymästä asiakkaan toimesta. Sovelluksessa ei saa olla virheitä ja käyttökokemuksen pitää olla hyvä. Autentikaatio, autorisaatio ja tietoturva pitää hoitaa hyvien käytänteiden mukaisesti.

Teknologioiden pitää antaa paljon vipuvoimaa kehitykseen. Teen opinnäytetyön kehitys ja raportointityötä töiden ja perheen hoidon ohella, eli aikaa on hyvin rajallisesti. Työaikaa saattaa olla vain kymmeniä minuutteja kerrallaan ja joka kerralla pitää saada merkittävää edistystä aikaan. Julkaisun pitää olla vaivatonta, eli onnistua yksittäisillä komennoilla.

Jatkokehitystä tullaan tekemään ja sen pitää olla tehokasta. Teknistä velkaa ei saa tietoisesti ottaa, vaan koko aika pitää tehdä parasta mahdollista jälkeä. Toiminnallisten muutosten yhteydessä on tehtävä myös ei toiminnallisia parannuksia, jos niitä on tehtävissä.

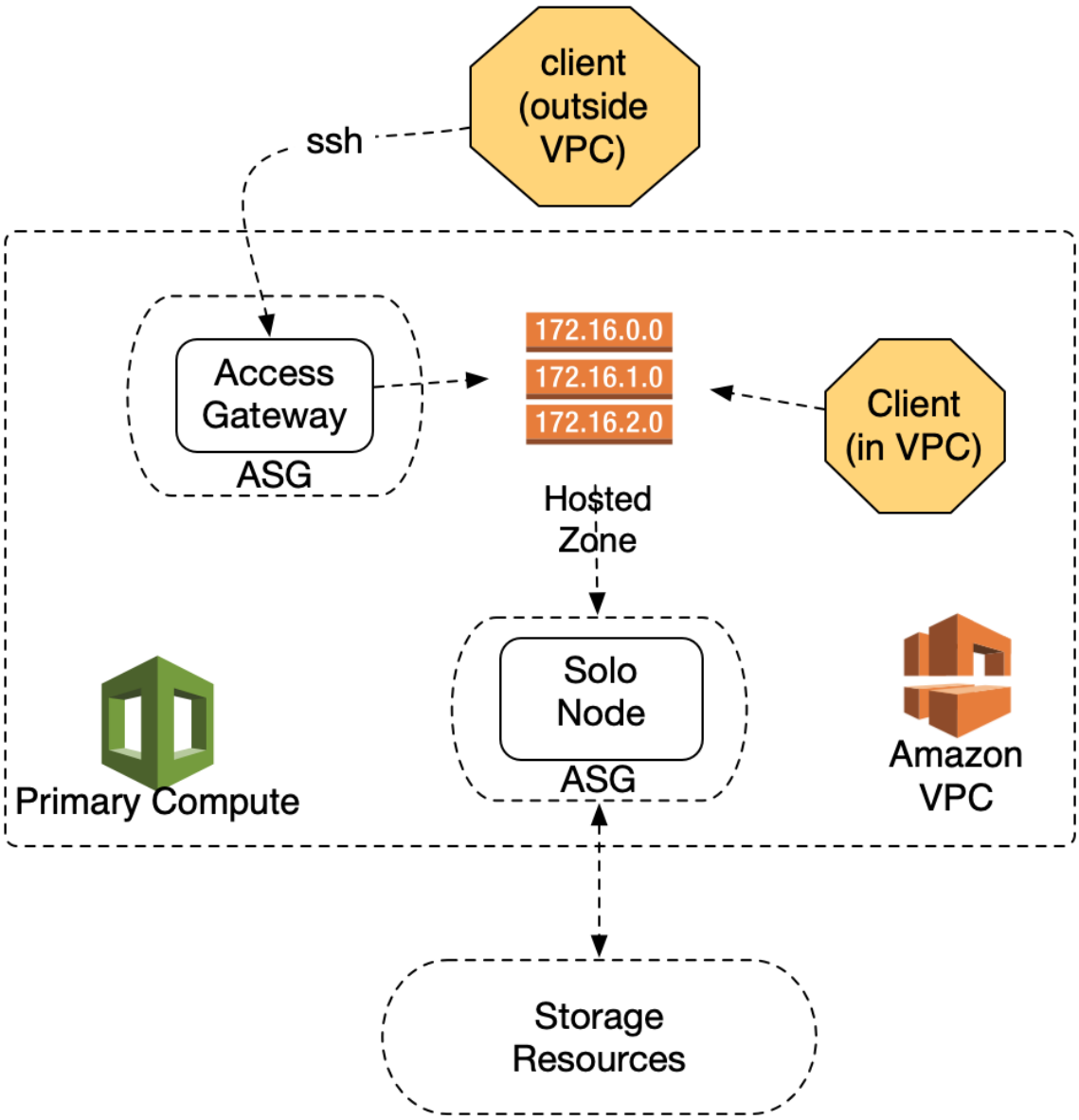
3.3 Arkkitehtuuri

Käyttöliittymä on selainpohjainen React.js -sovellus. Käyttöliittymä käännetään staattisiksi JavaScript paketeiksi, jotka ovat julkisesti saatavilla AWS S3 bucketista AWS CloudFront CDN (content delivery network) kautta. Käyttöliittymän avulla voi hakea rajapinnasta tokenin ja sen avulla voi tehdä muita rajapinta kyselyitä autentikoituneena.

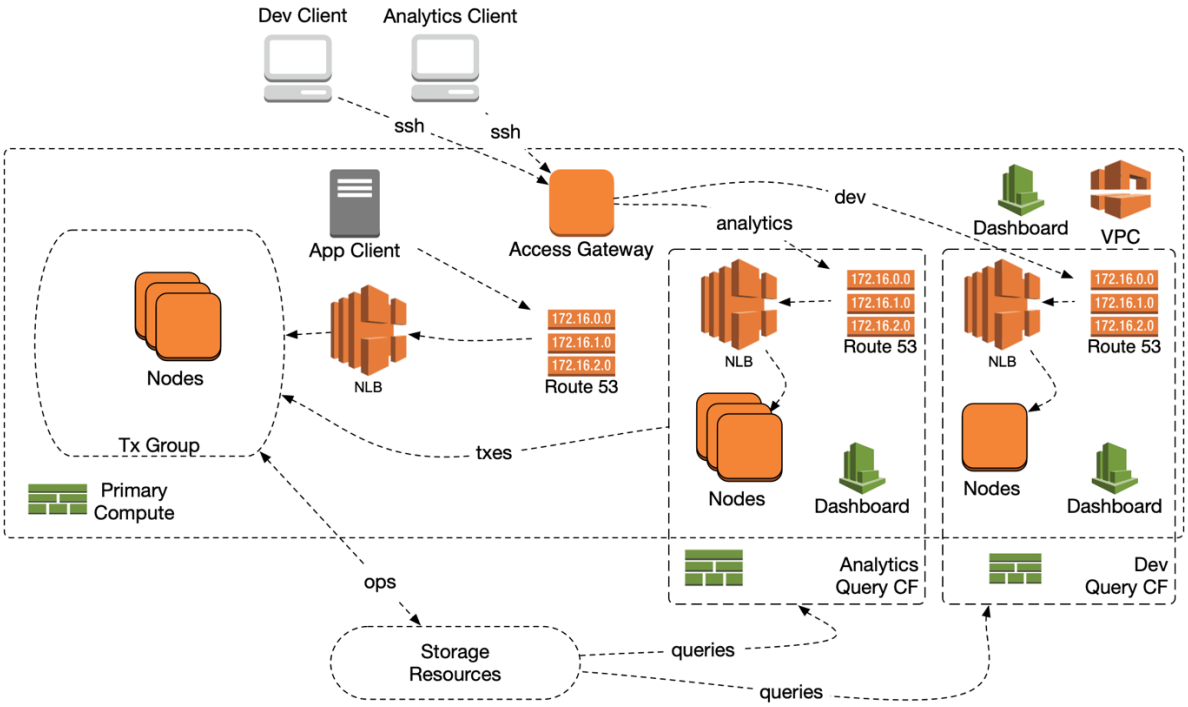
Rajapinta on julkaistu AWS API Gateway kautta ja yhdistyy AWS Lambda kautta Datomic Cloud Ioniin. Ioni on Clojure funktio, joka Ring-middlewarejen avulla parsii kutsun, dekoodaa valtuuden tiedot ja välittää kutsun tiedot EQL parserille. EQL parseri kutsuu oikeita resolvereita ja palauttaa kyselyn, sekä

käyttäjän perusteella oikean vastauksen. Resolverit tekevät suoraan tietokantakyselyitä Datomic tietokantaan, joka on samassa infrastruktuurissa kuin resolverit itse.

Tietokannan topologia on kehitysvaiheen versio Datomic Cloudista.



Se muutetaan tuotantovaiheen topologiaan tarpeen tullen ja tarkoituksen mukaisella laajuudella.



Tietokannan infrastruktuuri on koodina Cloudformation templaateissa, eli tietokantaa ei pystytetä manuaalisesti.

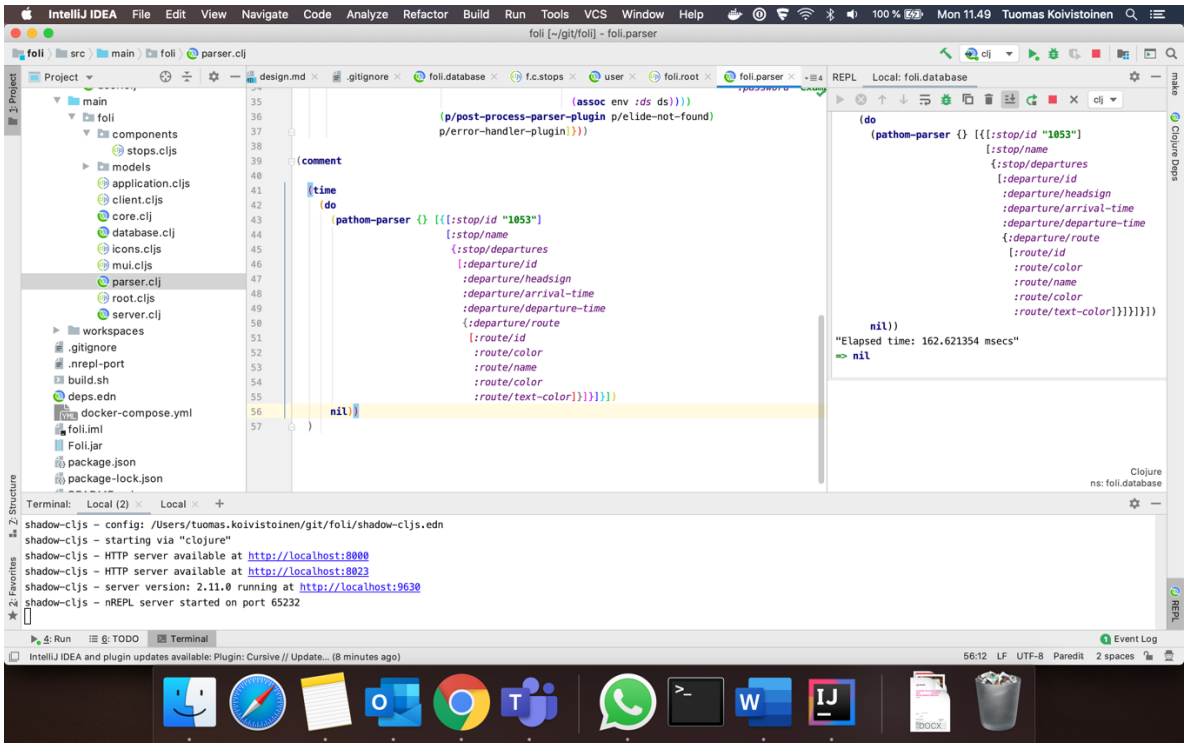
4 TEKNIIKKADEMONSTRAATIO

Ongelman ratkaisussa, suunnittelussa ja uusien ominaisuuksien tekemisessä olen kokeillut lähestyä sekä käyttöliittymän suunnalta tekemällä ensin käyttöliittymä prototyypin ja vasta sen jälkeen fullstack toteutuksen, että tekemällä ensin tietokantaan lisäykset, sitten rajapinnan ja lopuksi käyttöliittymän. Pidän molemmista tavoista, mutta ainakin loppuajasta suurin osa ominaisuuksista tuli tehtyä tietokanta ensin. Sovellus on salassapitosopimuksen alla, mutta demonstroin eri kehitysvaiheita toisten projektien tai keksittyjen esimerkkien avulla.

4.1 Editori ja REPL

Tässä kappaleessa näytetään esimerkki integroidusta kehitysympäristöstä ja kirjoitetaan funktio. Tarkoituksena on demonstroida minkälaista lähdekodiin editoiminen ja sen toiminnan varmentaminen on kyseisillä työkaluilla.

Sovelluksen kehitystä varten käytössä oli editorina Jetbrains IntelliJ IDEA ja Clojurea varten oli käytetty lisäksi Cursive Clojure lisäosaa. Clojurea voi kehittää millä tahansa tekstieditorilla, mutta tavanomaisesti käytetään editoria, jossa on REPL integraatio ja mahdollisuus rakenteelliseen editointiin. Yleisimmät editorit (ja pluginit/konfiguraatiot) ovat järjestäen Emacs (CIDER/Spacemacs), IntelliJ (Cursive Clojure), VS Code (Calva), Vim (vim-fireplace) ja Atom (Chlorine) (State of Clojure 2020).



Muissa kielissä on yleistä kohdella koodia merkkijonoina. Clojuressa koodia kohdellaan datana ja yleensä sitä myös editoidaan rakenteellisesti (structural editing). Muodostelmia (form) poistetaan, siirretään, niihin otetaan uusi jäsen, niistä poistetaan jäsen tai toteutetaan muita toimenpiteitä

näppäinyhdistelimen avulla. Editointi oli alussa vaikeaa, mutta sen opittua on vaikea palata editoimaan merkkijonona. Rakenteellinen editoiminen tuntuu paljon luontevammalta, vaikka sitä on tehnyt paljon vähemmän.

Esim. kuvitellaan funktio, joka jakaa luvun 42 argumentin luvulla.

```
(defn divide-42-by [x]
  (/ 42 x))
```

Virhetilanne voi syntyä, jos x on 0, joten tarkistetaan ettei niin ole. Kun avataan sulku, niin editori myös sulkee sen automaattisesti.

```
(defn divide-42-by [x]
  ()
  (/ 42 x))
```

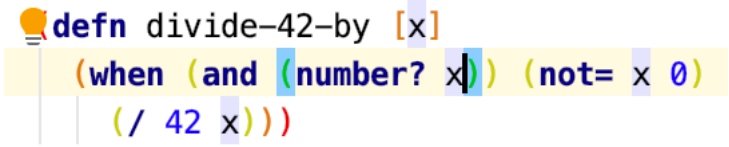
Palautusarvo halutaan vain silloin, kun x ei ole 0. Kirjoitetaan ehto loppuun.

```
(defn divide-42-by [x]
  (when (not= x 0))
  (/ 42 x))
```

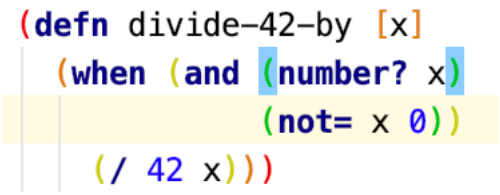
Editori sulkee formin sulun automaattisesti. Ongelmana on, että (/ 42 x) formi on saatava sen sisälle. Tätä varten voidaan käyttää rakenteellista muokkausta. Tässä tapauksessa voidaan ”slurpata eteenpäin” näppäinyhdistelmällä, eli ottaa seuraava formi nykyisen sisälle.

```
(defn divide-42-by [x]
  (when (not= x 0)
    (/ 42 x)))
```

Uusi ongelma voi syntyä, jos x ei ole numero. Lisätään alkuun tarkistus, että kysessä on numero. Hypätään näppäinyhdistelmällä kohtaan ennen ”valittua” (not= x 0) formia ja lisätään numero tarkistus.



Koska seuraava ”edessäpäin slurpattava” formi on (not= x 0) formi, niin näppäinyhdistelmä slurppaa sen. Lisäksi jaetaan ehdot eri riveille tuomaan selkeyttä.

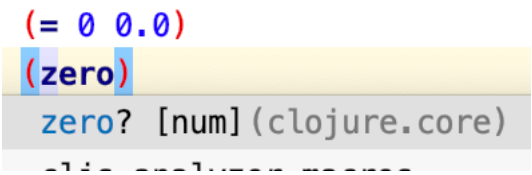


REPL-integraatiolla saa editoriin upotettuna REPL-ikkunan, jonne voi näppäinyhdistelmällä lähettää koodia evaluoitavaksi suoraan editorista. Lisäksi voi vaihtaa nimitilaa nykyiseen, ladata koodia tai esimerkiksi suorittaa mukautettuja toimintoja, kuten tiettyjä funktioita omilla näppäinyhdistelmillä. REPL toimii ohjelman kontekstissa, eli esim. ClojureScript REPL:issä voi suorittaa (js/alert ”Moi replistä”) ja nähdä varoituksen suoraan selaimessa. Vastaavasti Clojure REPL:issä voi tehdä kyselyitä tietokantaan tai rajapintoihin, käynnistää http-palvelimen uudestaan tai mitä tahansa muuta.

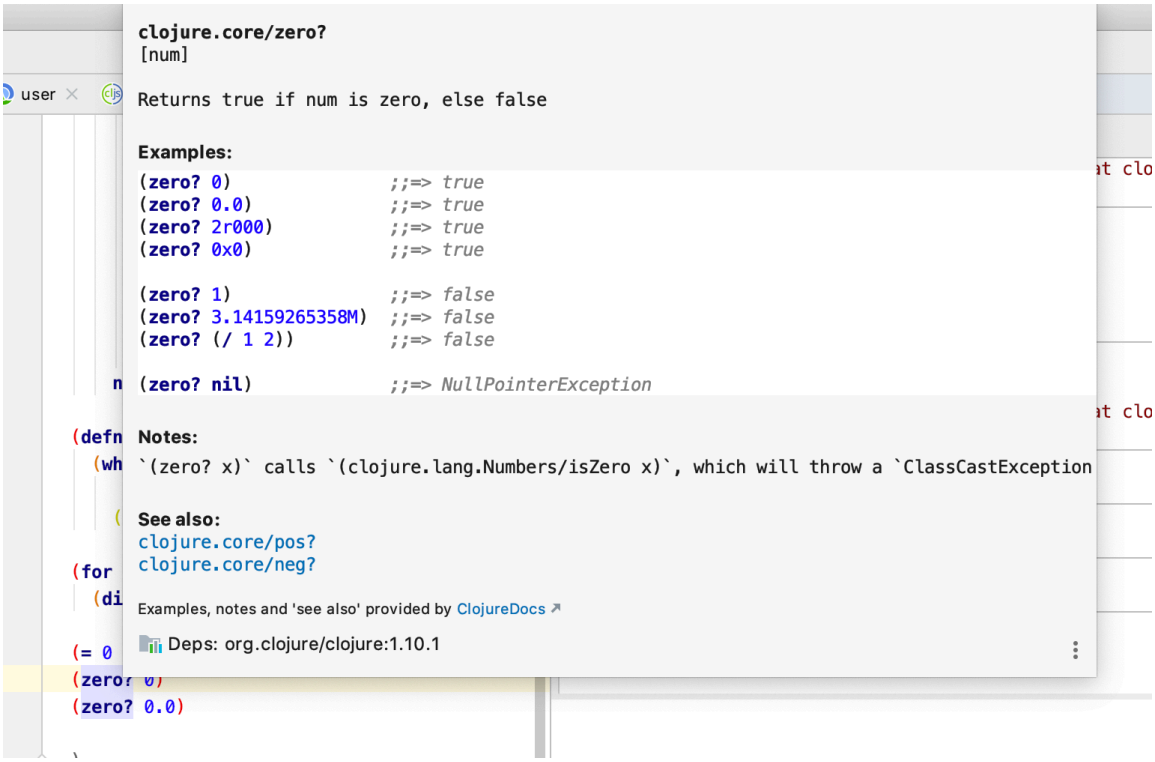
Esimerkkinä evaluoidaan suoraan editorista näppäinyhdistelmillä REPL:iin edellisen esimerkin funktio ja sen kutsuja.

```
(defn divide-42-by [x]
    (when (and (number? x)
               (not= x 0))
      (/ 42 x)))
=> #'foli.parser/divide-42-by
(for [x [0 0.0 "0" "tekstiä" 2]]
    (divide-42-by x))
Error printing return value (ArithmeticException) at clojure.lang.Numbers/divide (Numbers.java:188).
Divide by zero
```

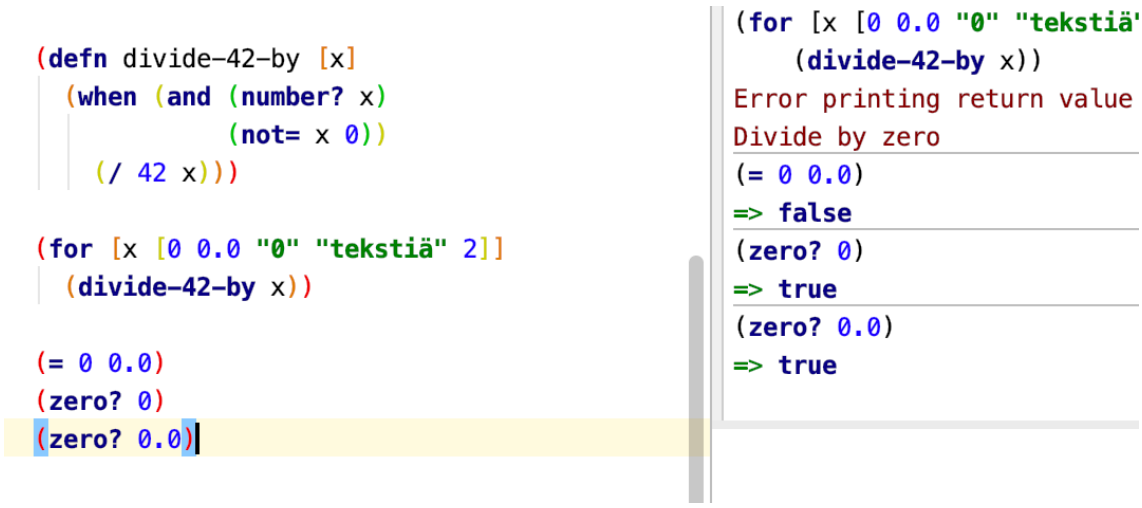
Eli toinen mahdollisuus virheeseen on desimaaliluku 0.0 jakaminen, joka on eri arvo kuin kokonaisluku 0. Kokeillaan löytääkö auto-complete valmista predikaattia.



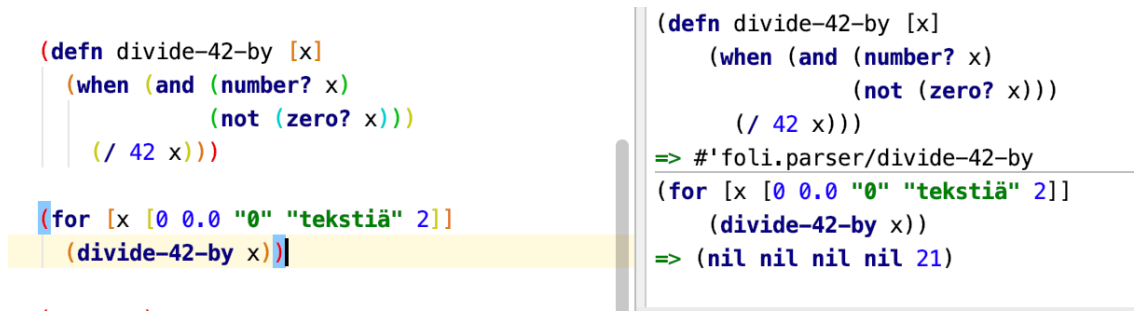
Luetaan dokumentaatio.



Evaluoidaan muutama esimerkkitapaus.



Ja lopuksi korjataan funktio.



Esimerkki on hyvin triviaali, mutta silti konkreettinen esimerkki siitä, miten Clojurella voi kehittää. Koodia voi evaluoida kirjoittamisen ohessa, askelia voi testata yksi kerralaan, askeleet voi yhdistää funktioksi ja lopuksi voi testata itse funktiota. Jos uskoo ratkaisevansa geeneristä ongelmaa esimerkiksi predikaattien tai kokoelmafunktioiden kanssa, niin voi etsiä ratkaisua Clojure Core -kirjastosta ja sen jälkeen muista kirjastoista.

4.2 Tietokannan kehitys

Tässä kappaleessa demonstroidaan Datomic kanssa työskentelyä. Tarkoituksena on demonstroida, minkälaista tämän tietokannan kanssa työskentely on. Demonstraation kaltaista työtä tehdään siinä kehitysvaiheessa, missä tietoa halutaan varastoida tai varastosta halutaan lukea tietoa.

Uusissa ominaisuuksissa on usein tarve uuden datan syöttämiselle, prosessoinnille ja näyttämiselle. Datan säilytystä varten on tietokanta ja uudenlaista dataa varten skeemaa pitää kasvattaa. Skeemaa vain kasvatetaan, eli mahdollisia attribuutteja vain lisätään. Jos attribuutissa on ongelma, joka joissain tietokannoissa voisi johtaa kolumnin poistoon, niin Datomic tapauksessa vanha deprekoidaan ja uusi tuodaan rinnalle. Clojuren yhteisö on vahva vakauden kannattaja ja rikkovien muutoksien vastustaja. Attribuutin poistaminen skeemasta olisi rikkova muutos.

Rikkovien muutoksien kieltäminen ja tiedon käsittely faktoina mahdollistaa paljon. Esimerkiksi iteratiivista tietokantamuutoksita ei ole tarvetta säilyttää migraatiotiedostoja koodin ohella versionhallintajärjestelmässä, koska muutoksista säilyy tieto tietokannassa itsessään. Ongelmien ratkaisuun saa myös lisää vipuvoimaa, koska jos tiedetään minä ajanhetkenä ongelma on tapahtunut, niin voidaan myös tietää mikä oli tietokannan tila samana ajanhetkenä. Tämä tietokanta voidaan kopioida ja sitä vasten voidaan suorittaa testejä.

Mielestäni paras tapa saada Datomic tietokanta käyttöön, on käyttää Cognitect julkaisemia CloudFormation-templaatteja. Ne voidaan suoraan ajaa omalle AWS-tilille ja saada parhaiden käytöntöjen mukainen ympäristö tietokannalle. Samaa ympäristöä voidaan myös käyttää sovelluskoodille ja funktioista voidaan ionien avulla tehdä rajapinta. Opinnäytetyön kehitysosuuden jälkeen heinäkuussa 2020 julkaistuun myös Datomic dev-local -kirjasto, jonka avulla voi kehittää ja testata Datomic sovelluksia täysin lokaalisti (Local Dev and CI with dev-local, 2020).

Jos on tehnyt Datomic ympäristön CloudFormation-templaattien avulla ja avannut sinne turvallisen yhteyden Datomic CLI kautta, niin ohjelmakoodissa voi luoda tietokanta clientin.

```
(def client (d/client {:server-type    :ion
                       :region         "enkerro"
                       :system         "enkerro"
                       :creds-profile "aws creds profile"
                       :endpoint       "enkerro"
                       :proxy-port     8182}))
```

Client avulla voi tehdä tietokannan ja ottaa siihen yhteyden.

```
(def db-name "testi123")
(d/create-database (get-client) {:db-name db-name})
(def conn (d/connect client {:db-name db-name}))
```

Yhteyden avulla voidaan tehdä transaktioida. Transaktiolla voidaan kasvattaa skeemaa tai viedä faktoja tietokantaan.

```
(d/transact conn {:tx-data [{:db/ident       :user/email
                             :db/valueType   :db.type/string
                             :db/cardinality :db.cardinality/one
                             :db/unique      :db.unique/identity}
                            {:db/ident       :user/password
                             :db/valueType   :db.type/string
                             :db/cardinality :db.cardinality/one}]})
```

Yhteyden avulla voi myös hakea muuttumattoman tietokannan. Tätä tietokantaa vastaan voidaan tehdä kyselyitä Datalog tai Datomic pull syntakseilla.

```
(d/q '[:find ?password
       :in $ ?email
       :where
       [?user :user/email ?email]
       [?user :user/password ?password]] db "tuomas.koivistoinen@edu.savonia.fi")
=> []
```

```
(d/pull db [:user/password] [:user/email "tuomas.koivistoinen@edu.savonia.fi"])
=> {}
```

Vaikka yhteyden läpi kulkisi transaktioita, niin haettu tietokanta on muuttumaton.

```
(d/transact conn {:tx-data [{:user/email    "tuomas.koivistoinen@edu.savonia.fi"
                             :user/password "salaamaton salasana"}]})
=>
{:db-before {:database-id "4be486b1-2c5e-4fb4-a655-303e19656e01",
             :db-name "testi123",
             :t 10,
             :next-t 11,
             :type :datomic.client/db},
 :db-after {:database-id "4be486b1-2c5e-4fb4-a655-303e19656e01",
            :db-name "testi123",
            :t 11,
            :next-t 12,
            :type :datomic.client/db},
 :tx-data [#datom[13194139533323 50 #inst"2020-11-04T09:36:03.043-00:00" 13194139533323 true]],
 :tempids {}}
(d/pull db [:user/password] [:user/email "tuomas.koivistoinen@edu.savonia.fi"])
=> {}
```

Jos halutaan nykyinen tila, niin haetaan nykyinen tietokanta.

```
(def db (d/db conn))
=> #'app.database/db
(d/pull db [:user/password] [:user/email "tuomas.koivistoinen@edu.savonia.fi"])
=> #:user{:password "salaamaton salasana"}
```

Sovelluksen kehityksessä uusien datan säilytystarpeiden ilmentyessä skeemaa voi kasvataa uusilla attribuuteilla, niillä voi lisätä testidataa ja tehdä testikyselyitä. Usein lopuksi samat testitransaktiot ja -kyselyt voi kopioida rajapinnan resolvereihin ja mutatioihin niihin sopiviksi muokattuina. Ongelmatilanteissa transaktioita ja kyselyitä pystyy muokkaamaan ja evaluoimaan suoraan lähdekoodin seassa ongelmien syiden selvittämiseksi.

4.3 Rajapinnan kehitys

Tässä kappaleessa demonstroidaan Pathom EQL -rajapinnan kanssa työskentelyä. Tarkoituksena on demonstroida, minkälaista tämän rajapinnan kanssa työskentely on. Demonstraation kaltaista työtä tehdään siinä kehitysvaiheessa, missä käyttäjältä halutaan ottaa vastaan tietoa, tai tietoa halutaan tarjota.

Uuden tiedon vastaanottamista varten tarvitaan mutaatioita ja tiedon tarjoamista varten tarvitaan resolvereita. Joskus uudet ominaisuudet ovat vain lisäyksiä olemassa oleviin, mutta joskus ne vaativat uuden tekemisen. Resolvereihin ja mutaatioihin pitää kirjoittaa tarvittu ja tuotettu data, sekä logiikka datan käyttöön ja tuottamiseen.

Esimerkiksi, jos käyttäjä haluaa pystyä tekemään tilauksia ja tilauksissa on tavaroita, niin rajapintaan voi tehdä mutaation tilauksen tekemiseen ja resolverin käyttäjän tilauksien hakemiseen tavaroineen. Rajapinnassa autentikoidaan käyttäjä ja onnistuneen autentikaation yhteydessä kutsutaan resolver ja

mutation funktioita sähköpostin, tietokannan ja yhteyden kanssa. Kun datamalli on selkeä ja skeemaan on lisätty attribuutteja, niin voi testata transaktion ja kyselyn tekemistä.

Kun logiikka on selvä, niin transaktiot ja kyselyt voi siirtää mutaatioihin ja resolvereihin muokaten ne niihin sopiviksi.

```
(pc/defmutation create-user-order [{:keys [email connection]} {items :order/items}]
  {::pc/params [{:order/items [:item/name]}]}
  (d/transact connection {:tx-data [{:order/user  [:user/email email]
                                     :order/items items}]}))

(pc/defresolver user-orders-resolver [{:keys [email database]} _]
  {::pc/output [{:user/orders
                 [:db/id
                  {:order/items [:db/id :item/name]}]}]}
  {:user/orders
   (-> database
       (d/pull [{:order/_user
                 [:db/id {:order/items
                          [:db/id :item/name]}]}]
               [:user/email email])
       :order/_user)})
```

Resolvereita ja mutaatioita saa testattua esimerkiksi tiedoston pohjan comment muodostelmassa, missä voi kutsua suoraan EQL parseria tyylillä (parser env [:user/name]). Env on ympäristö, jossa voi olla muun muassa käyttäjän valtuustiedot ja tietokantayhteys. Toinen argumentti on itse kysely. REPL-ikkunaan saa evaluoinnin tuloksen.

4.4 Käyttöliittymän kehitys

Tässä kappaleessa demonstroidaan Fulcron kanssa työskentelyä. Tarkoituksena on demonstroida, minkälaista tämän käyttöliittymäkirjaston kanssa työskentely on. Demonstraation kaltaista työtä tehdään siinä kehitysvaiheessa, missä käyttöliittymään pitää tehdä lisäyksiä.

Uusi ominaisuus vaatii usein lisäyksiä käyttöliittymään ja lisäystä varten voi olla tarpeellista tehdä uusi Fulcron tilaa käyttävä komponentti. Komponenttiin määritellään palvelimelta tarvittava data, käyttöliittymäkohtainen data, mahdollinen ident, logiikka datan hakemisen ajankohtaan ja joskus myös muita toiminnallisuuksia. Komponentin loppuun kirjoitetaan deklaratiivisesti renderöitävä näkymä parametrisoituna. Siinä voi hyödyntää React-komponentteja NPM tai Clojurescript ekosysteemeistä, suoraan HTML-elementtejä tai itse tehtyjä komponentteja.

Esimerkkinä käytän toista projektia, jossa voidaan bussipysäkin numerolla hakea tulevat lähdöt ja niiden lähtöaika, suunta ja linja. Tässä tarvitaan komponentti pysäkin valitsemiseen ja kun pysäkki on valittu, niin myös komponentteja muiden tietojen näyttämiseen. Pysäkin valintaan tarvitaan lomake, jossa on syötekenttä tekstille ja nappi lomakkeen lähetykseen.

Komponentissa kerrotaan mitä dataa se tarvitsee, mitä dataa halutaan lähtötilanteeseen, mikä on komponentin identifioiva attribuutti ja sen arvo, sekä mitä datan perusteella halutaan renderöidä.

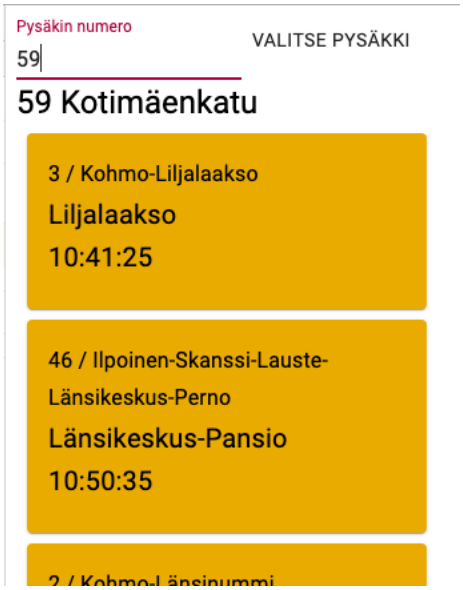
```
(comp/defsc StopSelection [this {:ui/keys             [input-text]
                                 :stop-selection/keys [stop]}]
  {:query         [:ui/input-text {:stop-selection/stop (comp/get-query StopDisplay)}]
   :ident         (fn [] [:component/id :stop-selection])
   :initial-state (fn [_] {:ui/input-text ""})}
  (dom/form
    {:onSubmit (fn on-stop-select [evt]
                 (.preventDefault evt)
                 (comp/transact! this [(select-stop {:stop/id input-text})]))}
    (mui/text-field
      {:label    "Pysäkin numero"
       :value    input-text
       :onChange (fn [evt] (m/set-string! this :ui/input-text :event evt))})
    (mui/button {:type "submit"} "Valitse pysäkki")
    (when (:stop/id stop)
      (ui-stop-display stop))))

(def ui-stop-selection (comp/factory StopSelection))
```

Komponentilla on nimi StopSelection, se tarvitsee toimiakseen propsina :ui/input-text, sekä :stop-selection/stop tiedot. Sen identifioiva attribuutti on :component/id ja sen arvo on vakio :stop-selection. Tämä tarkoittaa sitä, että komponentin data on käyttöliittymän tietokannassa :component/id taulussa :stop-selection arvolla aina. Renderöitäväksi tulee lomake, jossa on syötekenttä tekstille ja nappi

lomakkeen lähetykseen. Lisäksi jos on validi :stop-selection/stop arvo, niin se renderöidään loppuun. Syötekentän arvon muuttuessa asetetaan uusi :ui/input-text arvo näytettäväksi. Renderöitävät komponentit voisivat olla suoraan html elementtejä, mutta päätin hyödyntää NPM-ekosysteemistä Material UI React -kirjastoa ja sen komponetteja. Lomakkeen lähetyksessä estetään sivun uudelleen lataaminen, sekä transaktoidaan select-stop mutaatio ja sille annetaan tieto syötekentän nykyisestä arvosta.

```
(m/defmutation select-stop [{:stop/keys [id]}]
  (action
    [{:keys [app state]}]
    (swap! state assoc-in [:component/id :stop-selection :stop-selection/stop] [:stop/id id])
    (df/load! app [:stop/id id] StopDisplay
              {:update-query (update-key-params :stop/departures {:since (js/Date.)})})))
```

Select-stop mutaatiossa on vain action haara, jossa tehdään käyttöliittymän tason muutoksia. Annetun :stop/id arvon perusteella päivitään :component/id taulun :stop-selection entiteettiin :stop-selection/stop arvoksi viittaus toiseen entiteettiin, jonka identifioiva attribuutti on :stop/id ja arvo on annettu :stop/id arvo. Viittaus on tässä tapauksessa ident [:stop/id ”59”]. Pysäkin valinnan komponenttiin siis valitaan tietyn id:n omaava pysäkki. Tätä pysäkkiä ei välttämättä ole vielä käyttöliittymän tietokannassa ollenkaan, mutta valinnan lisäksi mutaatiossa myös ladataan kyseisen identin omaavan entiteetin data StopDisplay komponentin datamäärityksien perusteella. Ladattava data on siis tietyn pysäkin tietyt attribuutit. Mutaation argumentista saadaan tieto, että minkä pysäkin tietoa pitää ladata ja StopDisplay komponentin datamäärityksistä saadaan tieto, että mitä dataa pysäkistä tarvitaan. Tässä tapauksessa latauksen kyselyä myös päivitetään siten, että kyselyn :stop/departures attribuutin kohdalle annetaan myös :since parametri, jonka avulla voidaan rajata jo lähteneet lähdöt pois valmiiksi palvelimen puolella. Kun pysäkin tiedot on ladattu, niin ne voidaan näyttää.



Pysäkistä halutaan näyttää pysäkin numero, nimi ja sen lähdöt. Lähdöistä halutaan näyttää suunta, lähtöaika ja linja. Linjasta halutaan näyttää id ja nimi, mutta lisäksi halutaan linjakohtaiset väritiedot tekstille ja taustalle.

```
(comp/defsc StopDisplay [_ {:stop/keys [id name departures]}]
  {:ident :stop/id
   :query [:stop/id :stop/name
           {:stop/departures (comp/get-query DepartureDisplay)}]}
  (comp/fragment
   (mui/typography {:variant "h5"} id " " name)
   {}
   (->> departures
        (sort-by :departure/departure-time)
        (mapv (fn [departure]
                (ui-departure-display departure))))))
```

Pysäkin komponentin datamäärittelyihin tarvitaan tieto sen identifiointiin ja pysäkin tietojen renderöinnin datatarpeet. Pysäkki identifioidaan :stop/id attribuutilla, joka tulee komponetille ”propsina”, eli argumenttina. Datatarpeita on identifioivan attribuutin lisäksi myös pysäkin nimi ja pysäkin lähdöt. Renderöintiin halutaan otsikko, jossa näkyy pysäkin numero ja nimi, sekä järjestetetty lista pysäkin lähtöjä. Lähtöjen datatarpeet ja renderöinnin määrittelee lähtöjen oma komponentti.

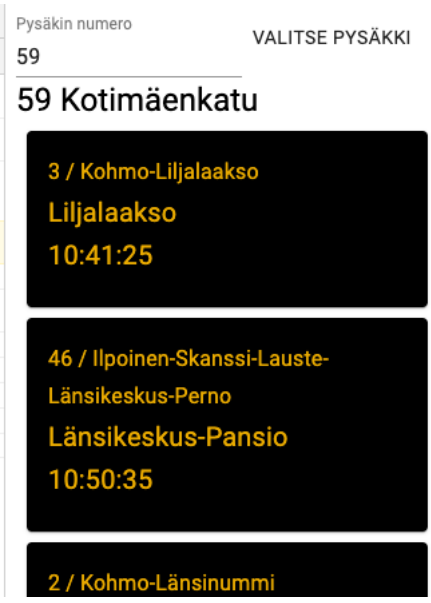
```
(comp/defsc DepartureDisplay [_ {:departure/keys [headsign departure-time route]}]
  {:ident :departure/id
   :query [:departure/id :departure/headsign :departure/departure-time
           {:departure/route (comp/get-query RouteDisplay)}]}
  (let [{:route/keys [color text-color]} route]
    (mui/card
     {:style {:margin          "8px"
              :maxWidth        "300px"
              :color           (str "#" text-color)
              :backgroundColor (str "#" color)}}
     (mui/card-content
      {}
      (ui-route-display route)
      (mui/typography {:variant "h6"} headsign)
      (mui/typography {:variant "h6"} departure-time)))))

(def ui-departure-display (comp/factory DepartureDisplay {:keyfn :departure/id}))
```

Lähdöistä tarvitaan suunta, lähtöaika ja linja. Linjan tiedoista voidaan ottaa tekstin ja taustan väri. Renderöitäväksi halutaan Material UI kortti, jossa näkyy linja, suunta ja lähtöaika. Kortin tekstin ja taustan väri saadaan linjan tiedoista.

```
(comp/defsc RouteDisplay [_ {:route/keys [id name]}]
  {:ident :route/id
   :query [:route/id :route/name :route/color :route/text-color]}
  (mui/typography {:variant "subtitle1"} id " / " name))

(def ui-route-display (comp/factory RouteDisplay))
```

Linjasta halutaan nimi, sekä taustan ja tekstin värit. Renderöitäväksi halutaan linjan id ja nimi.

Tavanomaisesti kehittäessä koodimuutokset halutaan ladata automaattisesti selaimeen ja tilaa ei haluta menettää. Tämä mahdollistaa todella nopean käyttöliittymän kehittämisen, koska kehitystä varten tarvittavan tilan joutuu luomaan vain kerran. Tilan säilyttävän koodin automaattisen selaimeen lataamisen ohella funktioiden testaaminen mielivaltaisilla arvoilla kätevällä näppäinyhdistelmällä suoraan editorissa on myös iso tekijä kehityksen mielekkyydessä, flow-tilan säilyttämisessä ja tehokkuudessa. Esimerkiksi, jos vaihdan koodissa kortin tekstin värin taustan väriksi ja taustan värin tekstin väriksi, niin tiedoston tallennuksen jälkeen muutos näkyy suoraan käyttöliittymässä ilman sivun lataamista tai tilan menettämistä.



5 YHTEENVETO

Olen tehnyt ohjelmistoja lyhyen uran aikana monilla eri tavoilla, ja jokaisessa tulee nopeasti raja vastaan tehokkuuden radikaalin kehittymisen suhteen. Ohjelmistokehitys tuntuu olevan kuin jongleerausta, jossa tehokkuus tietyllä teknologialla vaatii sitä, että pystyy jongleeraamaan useampaa palloa kerralla. Ongelmana on se, että maailmanennätyskin on vain 11 palloa, eli raja tulee nopeasti vastaan. Tehokkuutta saa lisää siten, että vaihtaa työkalut semmoisiin, joilla vähemmillä palloilla saa enemmän aikaan.

5.1 Clojure ja REPL

Ennen Clojurea suosin testilähtöistä kehitystä (test driven development), jotta saan mahdollisimman nopeasti palautteen koodin toimivuudesta. Nyt suosin Clojuren mahdollistamaa REPL-lähtöistä kehitystä. Palautteen koodin toimivuudesta saa paljon nopeammin, koodin voi suorittaa olemassa olevassa ympäristössä, eli ympäristöä tarvitse joka kerta alustaa uudelleen, koodia voi testata dynaamisesti tarpeen mukaan pienemmissä tai isommissa muodostelmissa ja REPL:in avulla voi vaikuttaa käynnissä olevan ohjelman toimintaan suoraan sitä uudelleen käynnistämättä. Opinnäytetyön kehittäminen oli todella mielekästä ja tehokasta. Isoin syy siihen oli REPL:in mahdollistama interaktiivinen ja dynaaminen kehitysmalli.

Opinnäytetyönä tehty sovellus on informaatiojärjestelmä ja Clojuren kannustama informaatiomalli on auttanut keskittymään toimialakohtaisiin ongelmiin keinotekoisten sijaan. Tarpeen tullen muut informaatiomallit ja ohjelmointiparadigmat ovat tuettuja kirjastojen avulla.

Sovelluksen toteutuksessa on hyödynnetty sitä, että Clojure on isännöity. Esimerkiksi käyttöliittymässä on käytetty suoraan Node Package Manager (NPM) ekosysteemin Victory-kirjastoa datavisuaalisaatioihin ja ClojureScriptillä kirjoitettu Fulcro käyttää Reactia. Palvelimen puolella on hyödynnetty suoraan esimerkisi java.util ja java.security paketteja salasanojen salaukseen. Sekä palvelimella, että käyttöliittymässä toimivia Clojure funktioita on kirjoitettu esimerkiksi salasanan validaatioon ilman koodin duplikoimista.

Opinnäytetyö ei olisi ollut toteutettavissa näin pienillä resursseilla ja isolla laajuudella ilman funktionaalista tyyliä. Koodikannan koko, testauksen vaatima aika ja muutoksiin kuluva vaiva ovat kaikki murto-osan siitä, mitä ne olisivat olleet imperatiivisella tyylillä. Virheet on ollut todella tehokas toistaa ja korjata, koska arvojen toistaminen on tilan toistamista helpompaa ja nopeampaa. Riippuvuuksien ja tilan minimoiminen tai kokonaan poistaminen on helpottanut järkeilyä.

5.2 ClojureScript, Shadow-cljs ja Google Closure Compiler

Opinnäytetyössä käytettiin lukuisia NPM-paketteja ja Shadow-cljs teki sen todella helpoksi. Shadow-cljs myös mahdollisti helpon Google Closure Compilerin hyödyntämisen. Olemassa olevaa JavaScript ekosysteemiä pystyy hyödyntämään täysin, lisäksi voi hyödyntää ClojureScript ekosysteemiä

vaivattomasti ja kaiken päälle saa Google Closure Compiler kehittyneet optimoinnit ClojureScriptistä käännetylle JavaScriptille.

NPM ekosysteemin helppo hyödyntäminen ja tilan säillyttävä koodin uudelleenlataaminen olivat isoja tekijöitä, jotka tehostivat kehitystyötä. Valmiilla NPM paketeillla sai paljon geneerisiä ongelmia ratkaistua suoraan ja käyttöliittymän hiominen oli todella miellyttävää, koska tila ei nollaantunut koodimuutosten yhteydessä.

Olen aikaisemmin käyttänyt muita työkaluja, kuten Webpack:ia, JavaScriptin optimointiin selaimia varten. Google Closure kääntäjä on huomattavasti edistyneempi, mutta edistyneet koko-ohjelma optimisaatiot vaativat erikoisen koodaustyylin. ClojureScriptin avulla voin kirjoittaa idiomaattista Clojurea ja saada kaupan päälle Google Closure kääntäjän edistyneille koko-ohjelma optimisaatioille yhteensopivaa JavaScriptia. ClojureScript ja sen tuomat ominaisuudet, kuten pysyvät ja muuttumattomat datarakenteet ovat taakka latausaikojen kannalta, mutta Google Closure kääntäjän ansiosta taakka on suhteellisen pieni ja projektin koon kasvaessa Google Closure kääntäjän tuomat hyödyt myös kasvavat.

React.js kanssa käytetään usein suorituskyvyn ja kehityksen tehokkuuden vuoksi Immutable.js ja Lodash paketteja. Google Closure kääntäjän ansiosta ClojureScriptillä saa vastaavat pysyvät ja muuttumattomat datarakenteet, sekä kätevän kirjaston datan käsittelyyn, mutta todella hyvin optimoituna koon ja suorituskyvyn kannalta.

5.3 Datomic tietokantana

Datomicin kyselyt ja transaktiot etäisesti muistuttavat SQL-kyselyitä ja -transaktioita, mutta ovat tiiviimpiä, deklaratiivisempia ja vain dataa. Datalogin ja etenkin pull syntaksin kanssa työskentely on ollut paljon mielekkäämpää kuin perinteinen SQL kyselyiden kirjoittaminen. Etenkin datalog säännöt ja suora datomic pull syntaksin käyttö tiedossa oleville entiteeteille ovat vähentäneet datan valitsemisen vaatimaa kognitiivista kuormaa ja mahdollistaneet suhteiden abstraktoinnin yleiskäyttöiseksi säännöksi.

Iso hyöty on myös se, että tietokannan infrastruktuurissa voi myös suorittaa sovelluskoodia ja sovelluskoodin funktioita voi julkaista rajapinnan kautta. Sovelluskoodin julkaisuun on valmiit työkalut, jotka helpottavat julkaisujen tekemistä huomattavasti. Korkeatasoisen tuotantojärjestelmän rakentaminen vaatii hyvin vähän vaivaa valmiiden CloudFormation templaattien ja muiden työkalujen ansiosta.

Säilytettävän tiedon käsittely faktoina antaa vipuvoimaa ongelmanratkaisuun. Sen sijaan, että tietokanta olisi varasto tiedolle, jossa tieto voi vapaasti muuttua tai kadota ilman jälkiä, niin se onkin kokoelma faktoja, mistä voi saada kyselyillä kuvan sillä hetkellä voimassa olevista faktoista, toisena ajankohtana voimassa olevista faktoista tai tietyn tiedon muuttumisesta suhteessa aikaan. On

hyödyllistä ajatella, että tietyn tiedon lisäämisen lisäksi myös sen poistaminen on fakta, joka tallennetaan.

5.4 EQL rajapinta

Koostava ja kyselypohjainen rajapinta mahdollistaa vaivattomasti kaiken pyytäjän tarvitseman tiedon palauttamisen yhdellä kyselyllä ilman ylimääräistä tietoa. Rajapinta ei palauta ylimääräistä tietoa ja tietoa ei tarvitse hakea usealla kyselyllä yksi toisen jälkeen. Tiedon haku on täten tehokasta ja rajapintaan ei tarvitse rakentaa ylimääräistä logiikkaa tämän mahdollistamiseen. Tarvittava tieto voi muuttua mielivaltaisesti ja tiedon palautus onnistuu yhtä tehokkaasti joka tapauksessa.

Toinen vaihto koostavalle ja kyselypohjaiselle rajapinnalle on Facebookin GraphQL. Itse koen GraphQL heikkoudeksi sen tyypit. Siinä missä EQL tapauksessa tiedon semantiikka on fieldissä ja täten fieldejä voi käyttää ja yhdistää kontekstisidonnaisesti, niin GraphQL tapauksessa tyyppi määrittelee fieldin semantiikan. Jotta yhden GraphQL tyypin fieldin saa resolvaamaan toiseksi tyypiksi, niin joko ensimmäisen tyypin resolverissa pitää hakea toisenkin tyypin data tai ensimmäisen tyypin fieldille pitää antaa oma tyyppi resolverinsa. Resolveri on usein vain funktio, jota voi tietenkin uudelleen käyttää, mutta mielestäni tämänlaisten ylimääräisten yhteyksien tekeminen hyvin alleviivaa tietokokoelmien tyypityksen aiheuttamia haastamia. EQL tapauksessa kokoelma tietoa on kontekstisidonnainen ja semantiikan saa suoraan fieldiin.

5.5 Käyttöliittymäkehitys Fulcrolla

Käyttöliittymä usein näyttää dataa, sen kautta voi tehdä erilaisia toimintoja ja se voi olla monessa eri tilassa. Fulcro tekee kaiken tämän määrittelystä ja hallinnasta järjestelmällistä. Komponenttien tarvitsema data voidaan määritellä itse komponettiin metadatana, erilaisten toimintojen aiheuttamat vaikutukset käyttöliittymässä ja palvelimella voidaan määritellä mutaatioissa, ja lisäksi tila on normalisoituna käyttöliittymän tietokantaan, josta sitä voi suoraan silmäillä ja josta se on saatavilla kaikkialla käyttöliittymässä.

Palvelinpuolella data säilötään normalisoituna tietokantaan ja sen voi saada tekemällä kyselyitä. Tietoa harvoin tallennetaan pelkästään lokaalisti muistiin eri funktioiden scopen sisälle, koska se voisi aiheuttaa saman datan monen eri version käytön samaan aikaan. Tieto voisi päivittyä yhteen paikkaan, mutta ei toiseen. Fulcron ansiosta myös käyttöliittymäpuolella saa datan säilöön normalisoituna tietokantaan ja se on saatavilla kaikille komponenteille datamäärityksien avulla aina ajantasaisena. Ei ole riskiä, että eri puolilla käyttöliittymää näytetään eri versioita samasta datasta ja ei ole tarvetta tehdä ad-hoc logiikka, jolla eri puolien tila päivitetään, kun yhdessä paikassa aiheutetaan muutos käyttöliittymän tilaan.

LÄHTEET JA TUOTETUT AINEISTOT

HICKEY, Rich 2020. A History of Clojure. [Viitattu 2020-11-01] Saatavissa: https://download.clojure.org/papers/clojure-hopl-iv-final.pdf

MOSELEY, Ben ja MARKS, Peter 2006. Out of the Tar Pit. [Viitattu 2020-11-01] Saatavissa: http://curtclifton.net/papers/MoseleyMarks06a.pdf

GOOGLE 2020. Closure Compiler. [Viitattu 2020-11-01] Saatavissa: https://developers.google.com/closure/compiler

MILLER, Alex 2020. State of Clojure 2020 Results. [Viitattu 2020-11-01] Saatavissa: https://clojure.org/news/2020/02/20/state-of-clojure-2020

COGNITECT 2020. Local Dev and CI with dev-local. [Viitattu 2020-11-25] Saatavissa: https://docs.datomic.com/cloud/dev-local.html

ORACLE 2020. GraalVM. [Viitattu 2020-11-25] Saatavissa: https://www.graalvm.org/