Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2019

Toni Huovinen

MONGO-TIETOKANNASSA TAPAHTUVIEN MUUTOSTEN VERTAILU VERSIOPÄIVITYSTEN YHTEYDESSÄ

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2019 | 42 sivua

Toni Huovinen

MONGO-TIETOKANNASSA TAPAHTUVIEN MUUTOSTEN VERTAILU VERSIOPÄIVITYSTEN YHTEYDESSÄ

Modernin ohjelmistokehityksen yksi kulmakivi on lähdekoodin ja tietokantojen muutosten seuraaminen. Ilman tätä mahdollisuutta ohjelmistoprojektien kehitys olisi huomattavan hidasta, ellei jopa mahdotonta. Opinnäytetyön tarkoituksena oli suunnitella BCB Medical Oy:lle kevyt ohjelma seuraamaan yrityksen Mongo-tietokannassa tapahtuvia dataobjektien muutoksia BCB:n asiakkaille julkaisemien verkkoselainpohjaisten ohjelmien toimituspäivitysten aikana. Luodun ohjelman oli tarkoitus toimia osana yrityksen käyttämää Atlassianin Bamboo-palvelua. Lähtötilanteessa vertailu suoritettiin manuaalisesti, mikä altistaa inhimillisille virheille.

Työ toteutettiin Java ja Groovy -ohjelmointikielillä IntelliJ IDEA -kehitysympäristössä. Muita työssä käytettyjä teknologioita olivat Mongo NoSQL -tietokanta ja tämän graafisena käyttöliittymänä Studio 3T, Sourcetree versionhallinnan työkaluna sekä Bamboo jatkuvan integraation ja toimituksen ympäristönä. Ohjelmakoodia lähdettiin rakentamaan iteroiden, mikä mahdollistaa nopeat testaukset ja korjaukset ongelmien ilmetessä.

Työn ensimmäinen versio täytti perusvaatimukset. Tämä toimi kuitenkin manuaalisesti eikä sen vuoksi ollut yhteensopiva Bamboon kanssa. Toista versiota lähdettiin kehittämään ensimmäisen pohjalta, ja tästä saatiin automaattisesti toimiva kokonaisuus. Tästä jatkokehitettiin versio, joka oli mahdollisimman yhteensopiva Bamboon kanssa, ja toiminnallisuuden vuoksi koodi jaettiin vielä kahtia. Ensimmäinen osa haki dataobjektien nykytilanteen ja tallensi sen JSON-muodossa tiedostoon. Toinen osa koodista luki aiemmin tallennetun tiedoston, haki dataobjektien päivitetyn nykytilanteen ja suoritti vertailun näiden kahden välillä. Tuloksena oli siisti raporttidokumentti Mongo-tietokantaan. Toimeksiannossa ei päästy aivan loppuun, skriptejä ei saatu vietyä mukaan varsinaisiin ohjelmiin virallisten tietokantayhteyksien uupuessa. Toimintaa kuitenkin pystyttiin simuloimaan onnistuneesti.

Johtopäätöksenä voidaan todeta, että Mongo-tietokantojen vertailun automatisointi osaksi jatkuvan toimituksen putkea on mahdollista. Tämä vähentää manuaalisen tarkistuksen määrää ja virheiden syntyä.

ASIASANAT:

MongoDB, Java, Groovy, Differentiointi

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and communications technology

2019 | 42 pages

Toni Huovinen

DIFFERENTIATION OF CHANGES IN MONGO DATABASE DURING SOFTWARE UDPATE PROCESS

One of the cornerstones in modern software development is the ability to track changes in your source code and database, without this ability developing new bigger software would be considerably more slower if not impossible. In this thesis report, I will guide the reader through the process in which I created a scripts for BCB Medical Ltd which are able to track the changes happening in their Mongo database regarding dataobjects during software update process. Program I was tasked to create was meant to be part of their Atlassian Bamboo continuous integration and deployment pipeline. Currently database changes are checked manually and that is slow and opens door to the possibility of errors.

In this thesis I am introducing the reader to the technologies and languages I used to create the script. These were Java and Groovy programming language and IntelliJ IDEA as an integrated development environment. For version control client I used Sourcetree and for managing Mongo databases I used Studio 3T. Bamboo was used as a platform for continuous integration and development. Studio 3T acted as a source of inspiration when I was designing the script. As for the creation of scripts, I chose an iterative approach for rapid testing and quick fixing.

First version of script fulfilled the basic needs that we agreed upon but it needed manual input from user and thus it was not suitable. This version gave a good base for the second version which I was able to turn into fully automatic script. For Bamboo I divided the script into two pieces, first part was responsible for fetching current dataobjects from the database and stored that data as a JSON file. Second part of script read the JSON file created earlier and fetched the current, possibly updated dataobjects from the database and performed the comparison between the two datasets. That resulted in a clean report document which was stored in Mongo database. Unfortunately, I wasn't able to put the scripts in the actual development environment due to the lack of official database connections. However, I was able to simulate functionality using JSON files in Bamboo and through IntelliJ IDEA using databases.

In conclusion, I am safe to say that it is indeed possible to create a script or two for the Bamboo environment and automate the differentiation process. Automation significantly speeds up the processes, releases people to do other tasks and lowers the possibilities of errors.

KEYWORDS:

MongoDB, Java, Groovy, Differentiation

SISÄLTÖ

1 JOHDANTO **7**

2 OHJELMISTOSUUNNITTELUN ELINKAARI **9**

2.1 Differentiointi 10

2.2 MongoDB-kokoelmien vertailu Studio 3T -ohjelmalla 11

3 KÄYTETYT TEKNOLOGIAT **12**

3.1 Kehitysympäristö IntelliJ IDEA 12

3.2 Java-ohjelmointikieli 12

3.3 Apache Groovy -ohjelmointikieli 13

3.4 Versionhallinta 15

3.5 Bamboo CI ja CD 15

3.6 MongoDB-tietokanta 16

3.7 Mongon ja MySQL:n eroavaisuuksia 17

4 SKRIPTIN OHJELMOINTI **19**

4.1 Johdanto tekemiseen 19

4.2 Suunnittelu 19

4.3 Ensimmäinen versio 20

4.3.1 Hello Groovy World! 20

4.3.2 Aloitus 21

4.3.3 Yhteys tietokantaan 23

4.3.4 Vertailu 24

4.3.5 Raportin luonti MongoDB-tietokantaan 26

4.3.6 Koodin ajon turvaaminen ja loppusilaukset 27

4.4 Toinen versio 28

4.4.1 Suunnittelu 28

4.4.2 Tietojen haku ja dataobjektien tarkastaminen 29

4.4.3 Vertailu 30

4.4.4 Raportin luonti MongoDB-tietokantaan 32

4.5 Skriptin vieminen Bamboo-palvelimelle 33

4.5.1 Ympäristön luonti ja testiajot 33

4.5.2 Skriptin sovitus osaksi putkea 35

5 TULOKSET **37**
5.1 Skriptin toiminta 37
5.2 Skriptien ajo Bamboo-palvelimella 38

6 YHTEENVETO **40**
6.1 Jatkokehitysajatuksia 40

LÄHTEET **42**

KUVAT

Kuva 1. Java- ja Groovy-esimerkit. Java vasemmalla, Groovy oikealla. 13
Kuva 2. Esimerkki each-metodista ja Closuresta. 14
Kuva 3. Esimerkki MongoDB-dokumentista. 16
Kuva 4. Esimerkki SQL- ja NoSQL-kielistä. Ylempänä SQL. 18
Kuva 5. Skriptin testaamista. 21
Kuva 6. Ensimmäinen etappi. 22
Kuva 7. Tietokantayhteys sekä kokoelmien haku. 23
Kuva 8. Haku, jolla pääsee dataobjektin kenttiin asti. 24
Kuva 9. Vertailu jolla selvitetään uudet ja poistuneet kentät. 25
Kuva 10. Kaksi esimerkkiä tulosteista. 26
Kuva 11. MongoDB-dokumentin luontia. 26
Kuva 12. Esimerkki muutosten raportoinnista MongoDB-tietokannassa. 27
Kuva 13. Uusien ja kadonneiden dataobjektien tarkistus. 30
Kuva 14. Uusi paranneltu vertailu. 31
Kuva 15. Map-rakenne dataobjektin tuloksille. 31
Kuva 16. Päivitetty raporttidokumentti. 32
Kuva 17. Esimerkki raporttidokumentista Mongo-tietokannassa. 32
Kuva 18. Bamboo-tehtävän toiminnot. 34
Kuva 19. Shell-komentoja testaamista varten. 34
Kuva 20. Bamboo-testiajon tulokset. 35
Kuva 21. JSON-tiedoston luominen. 36
Kuva 22. JSON-tiedoston lukeminen toisessa skriptissä. 36
Kuva 23. Bamboo-ajon tuloste vertailusta. 38

KÄYTETYT LYHENTEET JA SANASTO

CI ja CD	Jatkuva integraatio ja toimitus (Continuous integration and delivery).
Framework	Vapaasti käännettynä kehityskehys. Kehitysohjelmisto, joka tuo mukanaan esim. koodikirjastoja ja työkaluja, joilla jonkin ohjelmiston kehitys helpottuu.
IDE	Integrated Development Environment. Ohjelmointiympäristö, johon on usein sisällytetty ohjelmointia helpottavia työkaluja.
MVP	Minimum Viable Product, tuotteen prototyyppimalli joka täyttää suunnittelussa määritellyt vaatimukset. Käytetään usein pohjana varsinaiselle tuotteelle sekä tuotteen toimivuuden analysoinnissa.
Repositorio	Nimitys versionhallissa tietyn rakenteen mukaan luotuja datavarastoja.
SDLC	Software Development Lifecycle. Ohjelmistokehityksen elinkaari.
Skripti	Itsenäinen ohjelmakoodi, jonka voi ajaa esim. komentoriviltä. Skripteillä usein automatisoidaan toimintoja.

1 JOHDANTO

Erilaisten asioiden vertailua on tehty jo pitkään, atomitason muutoksista galaksien eroavaisuuksiin ja kaikkea siltä väliltä. Kuten jo sana vertailu ehdottaa, halutaan sillä erotella kaksi tai useampi elementti toisistaan perustuen joihinkin ennaltamäärättyihin parametreihin. Tietotekniikan maailmassa vertailuominaisuuksia löytyy esimerkiksi Word-ohjelmistosta, jossa voit helposti vertailla kahden asiakirjan välisiä eroja. Modernissa ohjelmistokehitysmaailmassa vertailu on yksi peruspilareista, jonka avulla saavutetaan nopea ja tehokas ohjelmiston kehittäminen. Kehittäjätiimi pysyy lähdekoodin muutosten perässä ja mahdolliset ongelmatilanteet pystytään jäljittämään projektin alkuun asti. Tietokantojen muutosten vertailussa varmistetaan, ettei asiakkaalle lähde mahdollisesti rikkinäistä versiota, jossa esimerkiksi jotain tietoa olisi kadonnut päivityksen yhteydessä. Tietojen oikeellisuus on tietenkin aina ykkösijalla, mutta tämä korostuu entisestään potilastietojärjestelmien kanssa toimiessa.

Tämän työn toimeksiantaja, BCB Medical Oy, tunnisti selkeän tarpeen heidän Mongo-tietokantojen muutosten vertailussa. BCB Medical Oy kehittää ja ylläpitää merkittäviä tautikohtaisia selainpohjaisia ohjelmistoja, joiden vakaana pohjana toimivat Mongo-tietokannat. Lähtötilanteessa ohjelmien päivitysten yhteydessä Mongo-tietokantojen data-objektien muutokset tarkastetaan manuaalisesti. Manuaalinen tarkastus on hidas prosessi ja altistaa inhimillisille virheille. Tarkistuksen ajan tiiminjäsen on pois muusta toiminnasta. Toimeksiannon tarkoituksena on luoda palvelu tai skripti, joka hoitaa Mongo-tietokannan muutosten vertailun automaattisesti osana ohjelman toimituspäivitystä.

Toimituspäivitys tehdään Atlassianin Bamboo-palvelulla, joten tavoitteena on, että lopullinen skripti on osa tuota Bamboo-putkea. Työ oli myös osaksi tutkimusta siitä, onko moinen automatisointi ylipäätään mahdollista.

Työ toteutettiin ohjelmoimalla vertailun suorittava skripti Groovy-ohjelmointikielellä IntelliJ IDEA -kehitysympäristössä. Työssä otettiin pohjaksi iteroiva lähestyminen. Tämä tarkoittaa sitä, että ohjelmakoodi rakennetaan pieni looginen pala kerrallaan. Iteroiva lähestymistapa mahdollistaa jatkuvan testaamisen ja nopeat korjaukset ongelmien ilmaantuessa.

Toimeksiantaja toivoo saavansa käyttöönsä ohjelmakoodin, joka suorittaa dataobjektien vertailun siten, että nähtäisiin uusien kenttien sekä poistuneiden kenttien tiedot. Yksi toimeksiannon keskeisimimmistä toiveista oli, että luotu ohjelmakoodi toimii osana toimituksen putkea ja tuloksista syntyy oma raportti Mongo-tietokantaan. Raportin avulla yrityksen analytiikkaosasto saa tietoa jatkotoimenpiteiden tekemiseen.

Tässä opinnäytetyössä käyn läpi työn kulun varsin yksityiskohtaisesti, näytän avainkohdat selkeillä kuvilla. Kuitenkin tätä ennen selitetään ohjelmistokehityksestä yleisesti sekä esittelen erään Mongo-tietokantojen käyttöön suunnitellun ohjelmiston vertailuominaisuuden. Käytin myös tätä vertailuominaisuutta inspiraation lähteenä suunnitellessani omaa toteutustani. Esittelen myös käytetyt teknologiat mm. Groovy-ohjelmointikielen sekä Mongo-tietokannan.

Mainitsen tässä opinnäytetyössä usein dataobjektit, jotka ovat työn toimeksiantajan pyynnöstä käyttämäni termi vertailun kohteille. Dataobjektit ovat Mongo-tietokantarakenteita, jotka pitävät sisällään ohjelman käyttäjälle näytettävän tiedon.

2 OHJELMISTOSUUNNITTELUN ELINKAARI

Jos tarkastellaan mitä tahansa ohjelmaa, sitä ei suinkaan ole alettu ohjelmoimaan suoraan ilman taustatyötä. ISO/IEC 12207 -standardi tarjoaa kattavat prosessikuvaukset ohjelmistokehityksen eri vaiheista, aina alkusuunnittelusta valmiin tuotteen ylläpitoon. Koska en tässä työssä varsinaisesti kehitä uutta ohjelmistoa, vaan teen skriptin joka toimii osana kokonaisuutta, käyn tässä kappaleessa vain kevyesti läpi ohjelmistosuunnittelun elinkaaren yleisellä tasolla. Kerron myöhemmissä kappaleissa toimeksiantokohtaisesti suunnittelusta ja toteutuksesta tarkemmin.

Ohjelmistoprojektin tulee alkaa aina kunnollisella vaatimusmäärittelyllä. Yleensä projektitiimin kokeneimmat jäsenet ovat mukana tässä prosessissa ja se suoritetaan yhteistyössä asiakkaan kanssa. Tästä yhteistyöstä syntyvää tulosta käytetään sunnitellessa lähestymistapaa ohjelmistoprojektin aloitukseen liittyen, esimerkiksi käytettävien teknologioiden suhteen tai selvitetään onko koko projekti ylipäätään mahdollista. [1,2]

Vaatimusmäärittelyn ja suunnittelun jälkeen aletaan tarkastelemaan syntyneitä määrittelyjä tarkemmin. Vaatimuksia tarkennetaan tarpeen mukaan ja luodaan määrittelydokumentti. Dokumentti vielä hyväksytetään asiakkaalla, ja sen jälkeen se toimii virallisena kehyksenä ohjelmiston kehitykseen koko sen elinkaaren ajan. [1,2]

Määrittelydokumentin avulla tuotesuunnittelijat kehittävät arkkitehtuurin, jonka avulla ohjelmisto kehitetään. Tästä syntyy suunnitteludokumentti, johon kirjataan mm. aikatauluja, budjettimäärittelyjä sekä mahdollisia riskejä. Dokumenttiin myös määritellään ohjelmistossa käytettävät mahdolliset tietokantayhteydet sekä ulkopuoliset integraatiot. [1,2]

Kun suunnitteludokumentti on valmis ja hyväksytetty asiakkailla sekä mahdollisilla sidosryhmillä, alkaa ohjelmiston varsinainen kehitys. Kehitystiimin on suositeltavaa noudattaa suunnitteludokumentin asettamia ohjeita sekä noudattaa yrityksen yhteisiä pelisääntöjä ohjelmointia suorittaessa. [1,2]

Yleistä testausta tapahtuu koko elinkaariprosessin ajan usein, mutta erityistä huomiota tulee kiinnittaa itse tuotteen testaukseen. Ohjelmistokehitysprojektin aikana kehitettyjä ominaisuuksia tulee testata järjestelmällisesti. Monet yritykset luovat kehitettävien ominaisuuksien kylkeen esimerkiksi yksikkötestejä, joilla varmistetaan uuden ominaisuuden toimivuus rajatapauksissa. Joskus ohjelmasta kehitetään versio, joka testataan ennen asiakkaalle testiin antamista ja esille nousseet virheet korjataan. [1,2]

Kun ohjelmistoa on saatu kehitettyä julkaisuversioksi asti ja testauksesta on tullut hyväksytty päätös, voidaan tuote julkaista asiakkaille. Ohjelmisto voidaan myös julkaista osissa ja täydentää myöhemmin versiopäivityksillä. Julkaisun jälkeen alkaa ylläpitovaihe, jonka pituus sekä yksityiskohdat ovat dokumentoitu määrittely- ja suunnitteludokumentissa. [1,2]

2.1 Differentiointi

Kun tietokantojen yhteydessä puhutaan differentioinnista, diffauksesta tai diffistä ylipäätään, tarkoitetaan usein tietokantarakenteiden vertailua keskenään. Vertailun avulla saadaan kattava kuva tietokannan muutoksista. Kun pysytään aktiivisesti muutosten perässä, pystytään mahdolliset ongelmakohdat löytämään nopeasti. [3]

Ajatellaan, että Mongo-tietokantaa käyttävä yritys julkaisee uuden versiopäivityksen ohjelmastaan, aivan kuten tämän työn toimeksiantaja tekee. Versiopäivityksen myötä ohjelman sisällä toimivat tietokantaa hyödyntävät näkymät saattavat muuttua rakenteellisesti. Jotta ohjelman uusi versio toimii oikein, pitää myös tietokantaa muistaa muuttaa rakenteellisesti siten, että se on yhteensopiva uuden version kanssa. Differentiointi koskettaa näitä muutoksia. Manuaalisesti, jos muutoksia lähtee tarkastelemaan, voi edessä olla mittava ja virheille altistava prosessi. Tässä kohtaa suureksi avuksi tulee kyseiset diff-menetelmät, jotka käyvät muutokset läpi nopeasti ja luovat tarvittaessa raportit löydöksistä. Raporteista selviää, ovatko tietokannan muutokset menneet läpi onnistuneesti ja voidaanko ohjelmiston päivitysprosessi viedä loppuun asti turvallisesti siten, ettei esimerkiksi tietoa katoa. [3]

Saatavilla on kaupallisia ohjelmia kuten Studio 3T, josta kerron seuraavaksi. Aina valmiit ohjelmat eivät toki sovi yrityksen tarpeisiin. Jos esimerkiksi muutosten tarkastelu halutaan osaksi jatkuvan integraation ja toimituksen putkea, ei kaupallista ohjelmaa välttämättä pysty liittämään osaksi tätä kokonaisuutta. Tällöin voidaan kehittää omia ratkaisuja, kuten olen tässä työssä tehnyt. Omat skriptit voidaan liittää osaksi putkea esimerkiksi Bamboo-ympäristössä.

2.2 MongoDB-kokoelmien vertailu Studio 3T -ohjelmalla

Studio 3T:n avulla pystytään vertailemaan Mongo-tietokantojen kokoelmia käyttäen graafista käyttöliittymää. Ensimmäinen vaihe on valita palvelimet, joissa vertailuun osallistuvat tietokannat ja näiden kokoelmat sijaitsevat. Tämän jälkeen käyttäjä voi raahata lähdetietokannan vertailutietokannan päälle ja kaikki samannimiset kokoelmat siirtyvät vertailuun automaattisesti. Käyttäjällä on myös vapaus valita kokoelmat manuaalisesti, eli näiden ei tarvitse olla samannimisiä. [3]

Kun käyttäjä on tyytyväinen valitsemiinsa kokoelmiin, voidaan vertailu aloittaa painamalla Run comparison -painiketta. Vertailun valmistuttua aukeaa raporttinäkymä, joka on jaettu kolmeen osaan, joista kaksi ensimmäistä ovat oleelliset perusvertailun kannalta. Ensimmäinen näkymä, yhteenveto, kirjaa ylös kaikki muutokset, joita dokumenteissa esiintyy. Toinen näkymä, eroavaisuudet, näyttää graafisesti kaikki ne muutokset, jotka analyysissä paljastuivat. [3]

Studio 3T esittää eroavaisuudet dokumenttitasolla sekä myös värikoodein. Vihreällä värillä näkyy ne dokumentit, jotka ovat ainoastaan lähteessä. Keltaisella värillä näkyvät dokumentit ovat sellaisia, jotka löytyvät molemmista kokoelmista, mutta sisältävät muutoksia toisiinsa nähden. Punaisella merkityt dokumentit ovat sellaisia, jotka löytyvät ainoastaan kohdekokoelmasta. Tulokset saadaan kätevästi ulos raportiksi CSV-muodossa. [3]

Tämä jaottelu oli mielestäni selkeä, joten päätin käyttää sitä inspiraationa vertailun suunnittelussa.

3 KÄYTETYT TEKNOLOGIAT

Tässä luvussa kerrotaan tarkemmin työssä käyttämistäni työkaluista ja ohjelmointikielistä. Ohjelmointiympäristönä käytössä oli IntelliJ IDEA:n Professional Edition. Pääasiallinen ohjelmointikieli on Groovy, mutta olen myös sivunnut hivenen Javaa, sillä Java toimii Groovyn tukevana pohjana. Versionhallinta on oleellinen osa mitä tahansa ohjelmistoprojektia ja kerron myös näistä tarkemmin. Toimeksiantaja käyttää jatkuvan integraation alustana Bamboota ja yksi tämän työn tärkeimmistä tavoitteista oli saada skripti osaksi Bamboo-ympäristöä. Tässä työssä keskeisessä osassa on MongoDB-dokumenttitietokanta, joten käyn myös sitä läpi.

3.1 Kehitysympäristö IntelliJ IDEA

IntelliJ IDEA on tšekkiläisen Jetbrains-nimisen yrityksen kehittämä ohjelmointiympäristö (IDE, Integrated Development Environment), josta on saatavilla kaupallinen versio sekä avoimeen lähdekoodiin perustuva ilmainen perusversio. IntelliJ IDEA julkaistiin ensimmäisen kerran tammikuussa 2001 ja se oli ensimmäisiä Java-ohjelmointiympäristöjä, joka tarjosi käyttäjälleen edistynyttä navigointia sekä koodin refaktorointia. Ohjelmointiympäristö nousi nopeasti kehittäjien keskuudessa suosituksi ja IntelliJ IDEA:n avoimen lisenssin versio on saatavilla jo useammalle kielelle, mm. Pythonille sekä PHP:lle. Googlen julkaisema Android Studio pohjautuu IntelliJ IDEA:n avoimen lähdekoodin versioon. [4]

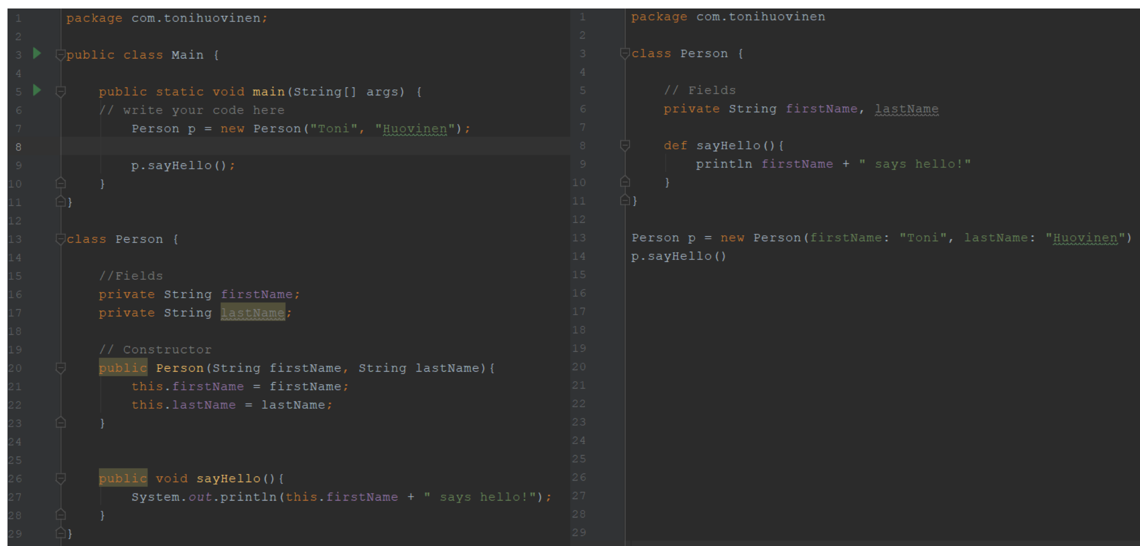
3.2 Java-ohjelmointikieli

Olio-ohjelmointikieli Java kehitettiin Sun Microsystems -nimisessä yrityksessä vuonna 1995. Oracle hankki itselleen Javan myöhemmin sekä sen kehittäneen yrityksen ja toimii nyt Javan kehittäjänä. Githubin ”Suosituimmat ohjelmointikielet” -listauksen mukaan Java on ollut yksi suosituimmista ohjelmointikieleistä jo pitkään, eikä ihme, sillä Java suunniteltiin alusta alkaen toimivaksi missä tahansa laitteessa. Javan suosion kantavana ajatuksena on ”write once, run everywhere”. Kirjoitettu koodi kääntyy bittikoodiksi, jonka voi ajaa millä tahansa Java-virtuaalikoneella huolimatta laitteen käyttöjärjestelmästä. [5]

3.3 Apache Groovy -ohjelmointikieli

Groovy on vuonna 2007 julkaistu monipuolinen ja dynaaminen olio-ohjelmointikieli, joka on täysin yhteensopiva Javan kanssa. Javaa voi kirjoittaa Groovyn joukkoon ja Java-ohjelmat voivat hyödyntää Groovy-koodia. Siinä, missä Java on ns. vahvasti tyypitetty kieli, Groovy sallii vaihtoehtoisen lähestymistavan. Tämä siis tarkoittaa, että muuttujat voidaan määritellä joko staattisesti tai dynaamisesti, esim. tekstin voi määritellä perinteisellä *String muuttujannimi* -tyylillä tai käyttäen avainsanaa *def muuttujannimi*. Groovy toimii samalla ”Write once, run everywhere” -periaatteella kuin Java. Se siis kääntyy bittikoodiksi ja sen voi ajaa millä Java-virtuaalikoneella vain. [6]

Groovyn eräitä vahvuuksia on se, että se pyrkii suoraviivaistamaan operaatioita, joita yleensä liittyy Java olio-ohjelmointiin. Esimerkiksi jotain luokkaa ohjelmoitaessa, tulee kirjoittaa asianmukaiset ominaisuudet, muodostimet, getterit ja setterit. Groovyllä vastaavaa luokkaa kirjoittaessa ei oikeastaan tarvita muuta kuin luokan ominaisuudet (Kuva 1), Groovy hoitaa taustalla kaiken tarvittavan. Tämä on tietenkin hyvin yksinkertainen esimerkki ja Groovy toki sallii ohjelmoijan kirjoittamat omat muodostimet ja metodit aivan kuten Java. [6]



Kuva 1. Java- ja Groovy-esimerkit. Java vasemmalla, Groovy oikealla.

Groovy tuo mukanaan myös liudan muita ominaisuuksia, joilla helpotetaan ohjelmoijan arkea. Näihin lukeutuu mm. tehokkaat metodit kokoelmien iterointiin sekä tehokkaat ehtolauseet. Iteroinnista käy hyvänä esimerkkinä listarakenteen läpikäynti. Usein käytössä on perinteinen for-silmukka, mutta Groovyllä ohjelmoitaessa voidaan käyttää lista-objektin mukana tulevaa each-metodia, joka hyödyntää Groovyn Closure-ominaisuutta (Kuva 2). Closuret ovat nimettömiä koodiblokkeja, jotka kirjoitetaan aaltosulkeiden sisään. Perinteisesti tapana on käyttää avainsanaa *it* esimerkiksi iteroitavien elementtien kanssa, mutta closureen voi syöttää minkä muuttujan nimen vain. Closuren sisällä muuttuja *it* saa aina senhetkisen arvon suoritettavasta koodista riippuen.

```
def lista = ['auto', 'pyörä', 'lentokone']
lista.each {println it}
```

Kuva 2. Esimerkki each-metodista ja Closuresta.

Groovy on siitä mielenkiintoinen kieli, että aivan kaikki luetaan olioiksi. Tämä siis tarkoittaa sitä, että kirjoitettaessa esimerkiksi

println 5.class

tulostaisi se tiedon siitä, että numero 5 on java.lang.Integer -luokan olio. Koska kaikki ovat olioita, tukevat ne luokillensa ominaisia metodeja. Yllä olevan esimerkin numero 5:stä saadaan helposti yksinkertainen looppi liittämällä siihen times-metodi, joka puolestaan taas toimii Closure-ominaisuutta hyödyntäen.

5.times { it ->…

Pythonin sekä Rubyn tavoin Groovy taipuu myös skriptikieleksi, tässä työssä pureudutaan tähän tarkemmin Groovyn suhteen.

Groovy on pääasiallisena ohjelmointikielenä Grails-frameworkissa. Grailsilla pystytään luomaan kohtalaisen vähällä vaivalla verkkosivuja, sovelluksia ja lisäosia. Työtä tehdessä minun olisi ollut mahdollista käyttää Grailsia, mutta työn tavoitteista johtuen Groovy-skripti toimii paremmin.

3.4 Versionhallinta

Ohjelmistokehityksessä on äärettömän tärkeää pysyä koodimuutosten perässä varsinkin, jos tekijöitä on useampia ja työ on mittava. Tähän tarkoitukseen on kehitetty erilaisia versionhallintaohjelmia, kuten esimerkiksi Git. Versionhallinnan avulla kehittäjät pystyvät tallentamaan kaikki koodimuutokset ja tarpeen vaatiessa palaamaan aikaisempiin versioihin, jos jotain menee pieleen.

Työn jakaminen osiin on helpompaa, sillä jokainen tiimin jäsen voi halutessaan tehdä muutoksia erillisiin tiedostoihin yhtäaikaisesti. Lopulta koodikanta tuodaan yhteen saumattomasti. Jotkin kehitystyökalut (IDE) pystyvät luomaan yhteyden versionhallintapalveluihin, jolloin itse ohjelmointityö suoraviivaistuu huomattavasti ja kaikki tarvittava hoituu yhdestä paikasta.

Versionhallintaa on perinteisesti käytetty komentorivin kautta. Syöttämällä komentoja voidaan mm. luoda uusia projektikohtaisia datavarastoja (repository) ja viedä sekä hakea koodimuutoksia. Nykyään tarjolla on myös kattava valikoima graafisia käyttöliitymiä versionhallinnan avuksi. Näiden avulla kehittäjä näkee visuaalisesti koko projektin historian sekä koodimuutokset rivitasolla. Tässä työssä käytössä on GitLab sekä graafisena käyttöliittymänä Sourcetree, joka on ilmainen Atlassianin kehittämä graafinen käyttöliittymä versionhallintaan.

3.5 Bamboo CI ja CD

Jatkuvalla integraatiolla ja toimituksella (engl. continuous integration and deployment) tarkoitetaan prosessia, jossa kehittäjätiimi tuottaa ohjelmistokoodia yhteiseen repositorioon monta kertaa päivässä. Nämä lisäykset tarkistetaan automatisoinnin avulla, jolloin mahdolliset virheet ja ongelmat jäävät prosessissa kiinni ja ne voidaan korjata aikaisessa vaiheessa. Koska lisäyksiä tehdään monta kertaa päivässä, ovat ne silloin myös helpommin hallittavissa kokonsa puolesta. [8]

Työn toimeksiantajalla on käytössä Atlassianin kehittämä Bamboo-niminen jatkuvan integraation ja toimituksen alusta. Bamboo on yhdistetty Jiraan sekä GitLabiin, jotka yhdessä luovat toimivan kokonaisuuden. Aina, kun joku tiimistä tuottaa koodia yhteiseen repositorioon, siitä lähtee viesti Jiraan ja sitä myötä kaikille tehtävää tarkasteleville ta-

hoille, jotka ovat projektissa mukana. Tieto uudesta koodista lähtee myös Bamboo-palvelulle, jossa tehdään mahdollisia testejä koodille. Tiimi pysyy näin jatkuvasti tietoisena siitä, mitä on tehty ja tarvittaessa koodimuutokset pystytään jäljittämään hyvin tehokkaasti yhdessä versionhallinnan avulla.

Bamboo-alustalle voidaan asettaa skriptejä ajettavaksi ns. triggerien avulla. Näin pystytään kustomoimaan omat prosessit hyvinkin tehokkaasti.

3.6 MongoDB-tietokanta

MongoDB on vuonna 2009 julkaistu NoSQL-tietokanta, joka on tehokas alustariippumaton tietokantamalli. Mongon toiminta perustuu dokumenttipohjaiseen ratkaisuun. Tietokannassa on kokoelmia (engl. collection), joiden sisällä olevat dokumentit koostuvat avain-arvo-pareista.

Rakenteellisesti dokumentti muistuttaa JSON-datarakennetta (Kuva 3). Dokumentti sisältää tietoa avain-arvo-parina ja arvona voi olla tekstiä, numeroita tai jopa taulukkoa sekä oliomuotoista dataa.

```
{
    "etunimi": "Matti",
    "sukunimi": "Meikalainen",
    "ika": 30,
    "puhnum": "123 456 7890",
    "osoite": {
        "katu": "Koulukuja 3",
        "kaupunki": "Turku",
        "postinum": 12345
    },
    "harrastukset": [
        "Jalkapallo",
        "Lenkkeily"
    ]
}
```

Kuva 3. Esimerkki MongoDB-dokumentista.

Usein dokumenttien sisällä näkee myös sisäkkäisiä dokumentteja, jotka sijaitsevat omissa kokoelmissaan ja joihin usein viitataan niiden omalla ObjectId-tunnuksella (_id-kenttä). ObjectId:n tarkoitus on tehdä jokaisesta dokumentista yksilöllinen ja aina, kun kokoelmaan talletetaan uusi dokumentti, MongoDB luo uuden ObjectId:n automaattisesti. Käyttäjä voi toki halutessaan luoda kokonaan oman tunnisteen, mutta usein päädytään käyttämään automaattisesti luotua ObjectId-tunnusta. [9]

ObjectId-tunnus on 12-tavuinen heksadesimaalikoodi, jossa ensimmäiset 4 tavua ovat aikaan liittyviä, seuraavat 3 tavua liittyvät koneen tunnistukseen, tämän jälkeen tulevat 2 tavua muodostuvat sen hetkisen prosessin tunnuksesta ja 3 viimeista tavua muodostuvat satunnaisesti. [9]

MongoDB:n vahvuuksia on sen nopeus ja kyky välttää tiukat skeemat. Siinä missä perinteisen relaatiopohjaisen SQL-tietokannan on pakko noudattaa tiettyä ennalta asetettua muotoa, Mongo-tietokantaan voi saman kokoelman sisään tallettaa toisistaan poikkeavia dokumentteja. Usein kuitenkin noudatetaan sääntöjä, joilla pidetään kokoelmat yhteneväisinä. [10]

3.7 Mongon ja MySQL:n eroavaisuuksia

Työn toimeksiantajalla on MongoDB:n lisäksi käytössä myös relaatiopohjainen MySQL-tietokanta, joiden yhteiskäyttöä kutsutaan usein hybridimalliksi. Vaikka työ koskee enemmänkin MongoDB-tietokantaa, on hyvä silti käydä kevyesti läpi myös relaatiopohjaista tietokantaa ja sitä, miten se eroaa MongoDB:stä. Alla olevassa taulukossa (Taulukko 1) on listattu hyvin lyhyesti näiden kahden tietokannan peruseroja.

Taulukko 1. Pieni ote MongoDB ja MySQL -tietokantojen ominaisuuksista.

Ominaisuus	MongoDB	MySQL
Alusta	Multiplatform	Multiplatform
Tietokannan skeema	Dynaaminen	Jäykkä
Tiedon muoto	Dokumentti	Taulukkomainen
Skaalautuvuus	Skaalautuu hyvin	Hankalaa

MySQL-tietokanta koostuu tauluista, ja taulut koostuvat riveistä ja sarakkeista. Rakenne muistuttaa oikeastaan hyvinkin paljon esimerkiksi Excel-taulua. Relaatiopohjaisessa tietokannassa, kuten MySQL:ssä, tieto jaetaan usein useaan erilliseen tauluun erilaisten riippuvuussääntöjen saattelemana sen sijaan, että kaikki tieto olisi yhdessä ja samassa taulussa. Taulujen tietoja voidaan lopulta yhdistää hakujen yhteydessä käyttäen erilaisia JOIN-komentoja primary- että foreign-avaimia hyödyntäen. [11, 12]

Koska MySQL:n tietokantarakenne on jäykkä, taulun jokaisen datarivin tulee sisältää samat tiedot. Jos jotain tietoa ei ole saatavilla, se korvataan usein jollain oletusarvolla, esimerkiksi NULL-arvolla. Mongo-tietokannassa tätä ongelmaa ei ole, kuten jo aikaisemmin kappaleen 3.6 loppupuolella totesin. Mongo-tietokanta sallii varsin vapaan dokumenttirakenteen. [11, 12]

Molemmilla tietokantyypeillä on oma tietokannan hallintaan liittyvä syntaksinsa. MySQL käyttää SQL-kieltä, joka tarkoittaa Structured Query Language. Mongolla puolestaan on käytössä Un-Structured Query Language, joka muistuttaa enemmän varsinaista ohjelmointikieltä pistenotaatioineen (Kuva 4).

```
2  INSERT INTO movies(movie_id, movie_name) VALUES (1, "007 Spectre");
3
4
5  db.movies.insert({
6      movie_id: 1,
7      movie_name: "007 Spectre"
8  });
9
```

Kuva 4. Esimerkki SQL- ja NoSQL-kielistä. Ylempänä SQL.

4 SKRIPTIN OHJELMOINTI

4.1 Johdanto tekemiseen

Tässä luvussa käyn läpi yksityiskohtaisesti skriptin luomisen suunnittelusta toteutukseen sekä raportoin myös mahdollisista ongelmatilanteista. Päädyin ottamaan skriptin ohjelmoinnissa lähestymistavaksi iteroivan mallin. Tällä tarkoitan sitä, että pyrin jakamaan ohjelmointiprosessin sopivan kokoisiin hallittaviin osuuksiin aloittaen projektin luomisesta GitLabiin ja päätyen tulosten esittelyyn.

Versionhallinnan apuvälineenä käytän Sourcetree-ohjelmaa. Tämä aikaisemmin esitelty sovellus on osoittaunut niin työtehtävissä kuin henkilökohtaisissa projekteissakin loistavaksi työkaluksi versionhallintaan. Sen graafisen käyttöliittymän avulla versionhallinnasta tulee helppoa, koska pystyn näkemään muutokset edellisiin versioihin nähden sekä tarvittaessa palaamaan aikaisempiin versioihin takaisin.

Kehitysympäristönä toimii IntelliJ IDEA ja ohjelmointikielenä Groovy. IntelliJ tarjoaa modernit koodin täydennykseen tarkoitetut työkalut sekä virheentunnistukset. IntelliJ:ssä on myös sisäänrakennettu komentokehoteikkuna, jolla voin ajaa skriptikoodin suoraan kehitysympäristön sisältä. Tämä yksinkertaistaa testaamista huomattavasti, koska kaikki oleellinen on yhdessä paikassa. Groovy puolestaan suoraviivaistaa itse ohjelmointia, kuten kyseisen kielen esittelykappaleessa totesin. Käytännössä tämän työn olisi voinut hoitaa millä tahansa skriptaukseen sopivalla kielellä, kuten Pythonilla. Groovy valikoitui siitä syystä, että se on linjassa toimeksiantajan käytössä olevien työvälineiden kanssa. Ohjelmat, joita työn toimeksiantaja kehittää, on ohjelmoitu käyttäen Grailsia sekä Groovy-ohjelmointikieltä.

4.2 Suunnittelu

Seuraavassa luetelmassa on kuvattuna alustava suunnitelma ensimmäisestä versiosta, jonka pohjalta lähden skriptiä kirjoittamaan. Lopputulos todennäköisesti näyttää hivenen erilaiselta, mutta tämä suunnitelma antaa hyvän lähtöasetelman.

- Käynnistetään skripti antamalla komentoriviargumentiksi muuttuvan dataobjektin tunnuskoodi.
- Grapella (@Grab) otetaan käyttöön ulkopuoliset kirjastot, esim. Gmongo.

- Haetaan tietokannasta alkuperäinen dataobjektin tilanne annetun tunnuskoodin perusteella.
- Otetaan talteen muuttujaan tämä tietokanta, def oldDb.
- Suoritetaan migraatioskriptin ajo.
  - Pyydetään käyttäjää ajamaan ohjelman konsolissa migraatioskripti, kuittaa painamalla enter.
- Otetaan talteen migraatioskriptin luoma päivitetty tietokanta, def newDb.
- Suoritetaan oldDb ja newDb -kannoille haku, jossa päästään mahdollisimman lähelle aluetta, jolla muutokset tapahtuvat.
  - Talletetaan syntyneet tietorakenteet omiin muuttujiinsa.
- Iteroidaan syntyneet tietorakenteet läpi.
  - Lasketaan uusien/poistuneiden kenttien lukumäärä per kysymyssetti.
    - Uudet kentät lkm – vanhat kentät lkm.
      - Jos arvo negatiivinen, silloin kenttiä poistuu, tunnistetaan tämä.
    - Tarkastetaan löytyykö päivitetystä kokoelmasta kenttää, jota ei löydy alkuperäisestä. Löydetty uusi kenttä tulostetaan.
    - Tarkastetaan löytyykö alkuperäisestä kokoelmasta kenttää, jota ei löydy päivitetystä. Löydetty poistuva kenttä tulostetaan.
  - Luodaan muutoksista järkevä dokumentti tietokantaan.

4.3 Ensimmäinen versio

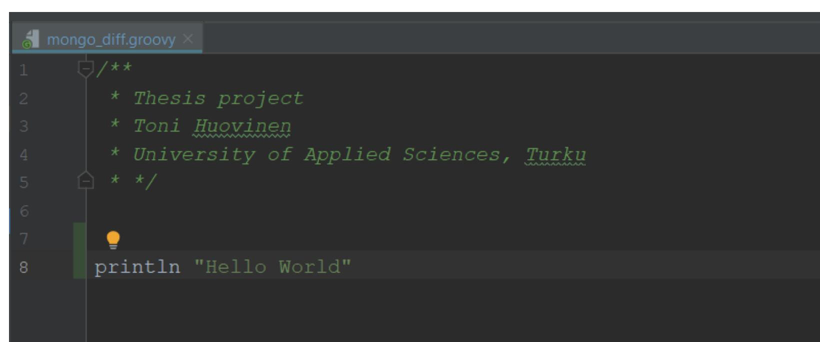
Ensimmäiseksi versioksi sovittiin toimeksiantajan kanssa sellainen skripti, joka suorittaa dataobjektin vertailun käyttäjän avustamana. Tämä tarkoittaa siis sitä, että skriptin käynnistyttyä käyttäjän tulee ilmoittaa tietokannan nimen sekä muutoksen kohteena olevan dataobjektin tunnus. Skripti toimii pohjana tuleville versioille ja täyttää MVP:n roolin.

Ennen kuin aloin kirjoittamaan ainuttakaan riviä koodia, piti selvittää versionhallinnan suhteen projektin sijainti. Sain luvan luoda tälle toimeksiannolle oman kansion yrityksen erääseen GitLab-repositorioon, minkä myötä tämän työn tulokset jäävät talteen ja voin tarvittaessa palata aiempiin iteraatioihin vakavampien ongelmien sattuessa. Bamboolla voidaan ottaa yhteys repositorioon ja mikäli skriptiin tulee tulevaisuudessa muutoksia, päivittynyt skripti on silloin saumattomasti käytössä.

4.3.1 Hello Groovy World!

Projektia aloittaessa on aina hyvä tehdä jokin yksinkertainen toiminnallisuus, jolla varmistetaan, että ohjelman voi ylipäätään ajaa ja toimiiko kehitysympäristö oikein. Halusin testata ajautuuko Groovy-skripti sellaisenaan komentorivillä, joten tein alla olevan (Kuva

5) hyvin yksinkertaisen skriptin, jonka tehtävänä on tulostaa perinteinen Hello World -teksti komentoikkunaan.



Kuva 5. Skriptin testaamista.

Yleisesti skripti ajetaan komentokehotteessa siten, että ensin kirjoitetaan se kieli, jolla skripti on kirjoitettu ja tällä tavoin valikoituu oikeat työkalut skriptin ymmärtämiseen. Sen perään kirjoitetaan skriptin nimi ja mikäli skripti hyödyntää komentoriviargumentteja voidaan ne kirjoittaa skriptin nimen perään. Tässä tilanteessa yllä oleva skripti ajautuu kirjoittamalla:

groovy mongo_diff.groovy

Tiedostopäätteen voi myös jättää pois. Skriptiä luodessa ja testattaessa pystyn ajamaan koodin myös IntelliJ:n kautta painamalla Suorita koodi -painiketta.

4.3.2 Aloitus

Testit on suoritettu ja on aika aloittaa ensimmäisen version kirjoittaminen. Ensimmäinen etappi, jonka haluan saavuttaa pitää sisällään ulkopuolisen Gmongo-kirjaston hakemisen sekä komentoriviargumenttien testaamisen. Komentoriviargumenttien avulla haluan luoda mahdollisuuden siihen, että skriptiä käynnistäessä voidaan antaa dataobjektin muutokseen liittyvä Jira-tehtävänumero sekä dataobjektin nimi. Myöhemmin lisäsin argumentteihin vielä mahdollisuuden antaa tietokannan nimen.

Varsinaisissa ohjelmistoprojekteissa riippuvuuksien hallinnassa käytetään usein esimerkiksi Mavenia tai Gradlea. Nämä ovat projektin rakentamiseen liittyviä automaatiotyökaluja. Kerron niistä seuraavaksi hyvin kevyesti, sillä ne eivät liity varsinaisesti tähän toi-

meksiantoon. Gradlella tai Mavenilla hallitaan mm. ulkopuolisten koodikirjastojen ylläpitoa. Ulkopuolisia koodikirjastoja nimitetään usein vapaasti suomennettuna riippuvuuksiksi. Gradlen avulla saat projektiisi uusimman version tarvittavasta kirjastosta kirjoittamalla gradle-tiedostoon kyseisen kirjaston nimen sekä versionumeron.

Miten tämä liittyy skriptiin? Koska skripti on oma erillinen ohjelmansa, sillä ei ole käytössä Gradlea tai Mavenia. Jollain tavalla ulkopuoliset kirjastot kuitenkin täytyy saada mukaan. Groovy hoitaa tämän käyttäen Grapesia. Grapes hakee Mavenin repositoriosta ne koodikirjastot, joita halutaan hyödyntää ja lisää ne ajon aikana luokkapolkuun mukaan.

Grapesin käytöstä on esimerkki alla olevassa kuvassa (Kuva 6). Grapes käynnistetään komennolla @Grab(…) ja sulkujen sisään laitetaan ryhmä, josta moduuli haetaan, moduulin nimi sekä moduulin versio.

```
/**
 * Thesis project
 * Toni Huovinen
 * University of Applied Sciences, Turku
 * */
@Grab(group = "com.gmongo", module = "gmongo", version = "1.3")

import com.gmongo.GMongo

def gmongo = new GMongo()


def version = args ? args[0]                                : "Oletusversio"
def objectName = (args && args.size() > 1) ? args[1]       : "Oletus nimi"

println version
println objectName
```

Kuva 6. Ensimmäinen etappi.

Yllä olevassa kuvassa on ensimmäinen etappi tavoitettu. Muutamassa kohdassa on näkyvissä punaista ja tämä on yleensä IDE:n tapa kertoa ongelmista. Nyt täytyy tosin muistaa ohjelman olevan skripti, joka siis lisää tarvittavat kirjastot mukaan ajon aikana ja tämän seurauksena nämä näkyvät virheet ovat siis vääriä hälytyksiä.

Version, joka siis on oikeastaan Jira-tehtävänumero, ja dataobjektin nimen otan talteen args-taulukkoon. Version-muuttujan suhteen testaan ensin onko args-taulukko ylipäätään olemassa eli se ei ole null. Jos se on olemassa, annetaan versiolle arvoksi kyseisen taulukon 0-indeksissä oleva arvo. Jos args-taulukkoa ei ole olemassa, saa versio arvoksi Oletusversio tekstin. Tämä rivi koodia noudattaa seuraavaa logiikkaa:

Jokin ehto ? Saa tämän arvon jos ehto on totta : Saa tämän arvon jos ehto ei ole totta

Dataobjektin nimen asettamisessa noudatan samaa ideaa, mutta teen vielä ylimääräisen tarkistuksen liittyen args-taulukon kokoon.

Lopulta tulostan ulos version ja dataobjektin nimen testatakseni niiden toimivuuden. Yksi useimmin käytetyistä debuggaus-menetelmistä, varsinaisen debuggaus-työkalun käytön lisäksi, on tulostaa ulos arvoja erilaisista kohdista. Tulosteiden avulla näkee helposti, mitä tapahtuu tai eteneekö suorittava koodi tulostukseen asti ylipäätään. Nyt, kun komentoriviargumentit ovat mukana, voin ajaa ohjelman seuraavasti:

groovy mongo_diff ABC-1234 JokuDataObjekti

Argumenttien syöttö onnistuu ja voin halutessa ajaa koodin myös IntelliJ:n kautta, jolloin versio ja dataobjektin nimi saavat niille määrätyt oletusarvot.

4.3.3 Yhteys tietokantaan

Seuraava etappi oli luoda yhteys tietokantaan, hakea sieltä oikea dataobjekti ja päästä kyseisessä rakenteessa sopivalle tasolle sen sisällön suhteen. Tätä ennen kuitenkin luon itselleni lokaaliin varastoon Mongoon thesis-nimisen tietokannan, jonka sisään kopioin kaksi olemassa olevaa kokoelmaa ja nimeän ne uudelleen Dataobjekti_orig sekä Dataobjekti_updated. Dataobjekti_orig pitää sisällään objektit alkuperäisessä muodossaan ja Dataobjekti_updated pitää sisällään samat objektit pienin muutoksin. Skriptin on tarkoitus havaita nämä muutokset alkuperäisen ja päivitetyn dataobjektin välillä.

Tietokannan sekä kahden kokoelman luomisen jälkeen lähdin kirjoittamaan koodia, jolla saan luotua yhteyden edellämainittuihin elementteihin. Tässä käytän apuna Grapesilla haettua GMongo-kirjastoa ja sen getDB-metodia (Kuva 7). Metodi ottaa sisäänsä tietokannan nimen tekstimuodossa. Muuttujaan db tallentuu tieto tietokannasta ja tämän avulla lopulta haen aiemmin luodut kokoelmat.

```
22
23      def db = gmongo.getDB("thesis")
24
25      // Get Collections
26      def originalCollection = db.Dataobject_orig
27      def updatedCollection = db.Dataobject_updated
28
```

Kuva 7. Tietokantayhteys sekä kokoelmien haku.

Koska nyt minulla on pääsy tietokantaan sekä kokoelmiin, pitää seuraavaksi mennä rakenteessa hivenen syvemmälle, varsinaisen vertailussa olevan dataobjektin kohdalle. Jokaisesta dataobjektista löytyy ylätasolta yksilöllinen koodi, jolla ne erotetaan toisistaan, joten tämä osoittautui hyväksi ankkuriksi haulle. Haku onnistui normaalilla MongoDB-haulla, findOne-metodilla. Kyseinen metodi ottaa kokoelman ensimmäisen dokumentin käsittelyyn ja hakukriteeriksi metodin sulkujen sisään laitoin avaimen johon dataobjektin yksilöivä koodi on sidottu ja tämän arvoksi skriptin käynnistyksen yhteydessä ilmoitetun objektin nimen (Kuva 8). Tarkensin hakua vielä entisestään muutamalla lisäominaisuudella, jotka jokaisesta dataobjektista löytyy.

```
27  // Get Collections
28  def originalCollection = db.Dataobject_orig
29  def updatedCollection = db.Dataobject_updated
30
31
32  // Focus on correct dataobject
33  def originalObject = originalCollection.findOne("template.code":objectName).template.questionSets
34  def updatedObject = originalCollection.findOne("template.code":objectName).template.questionSets
35
```

Kuva 8. Haku, jolla pääsee dataobjektin kenttiin asti.

Tällä haulla pääsin tietokannassa juuri siihen kohtaan, mihin halusin. Samalla paljastui ongelma. Rakenne, jonka haulla sain, osoittautui erittäin monimutkaiseksi ja syväksi. Dataobjekti hajautui lista-elementtiin, jonka sisällä oli map-elementtejä (Javan versio dictionary-datarakenteesta), joiden sisällä lista-elementtejä ja niin edelleen. Kävi ilmi hyvin pian, että kyseisen rakenteen läpikäynti ei ole ihan niin suoraviivaista mitä toimeksiannon ohjelmointi on tähän asti ollut.

Lähdin kokeilemaan erilaisia iterointikeinoja ongelmaan ja mistään ei tuntunut olevan apua. Koska kyseessä on skripti, jonka on tarkoitus toimia millä tahansa dataobjektilla, en voinut kirjoittaa tähän tilanteeseen spesifiä ratkaisua. Olin tämän pulman kanssa jumissa miltei yhden kokonaisen päivän, kunnes löysin Stackoverflowsta [13] oivan rekursiivisen metodin vastaavan rakenteen läpikäyntiin. Muokkasin metodia omiin tarkoituksiini sopivammaksi, ja se iteroi rakenteen läpi loistavasti.

4.3.4 Vertailu

Nyt kun rakenteen iterointi onnistuu, aloin pohtimaan parasta tapaa vertailla dataobjektin muutoksia. Tietokantahaussa käytin objektikohtaista koodia löytääkseni oikean paikan

ja tulin lopulta siihen tulokseen, että myös tässä kohtaa dataobjektien kenttäkoodien vertailu on järkevää. Dataobjektin jokaisella kentällä on oma yksilöllinen koodi, joten uusien kenttien ilmetessä tai vanhojen poistuessa muutokset heijastuvat kenttien koodiarvoihin.

Tässä on tosin yksi huono puoli. Jos dataobjektin kenttä muuttaa nimeään eli kentän otsikko vaihtuu, nämä muutokset eivät näy vertailussa. Tiedostin tämän puutteen jo tässä kohtaa ja päätin, että jatkan kenttäkoodien vertailulla ja sisällytän kenttien nimen muutokset seuraavaan versioon.

Aiemmin löytyneellä metodilla kenttäkoodien saaminen rakenteesta oli helppoa ja erottelin sekä alkuperäisestä objektikoelmasta että päivitetystä objektikokoelmasta kenttäkoodit erilleen omiksi listoiksi. Näiden listojen vertailu oli huomattavan helppoa, koska Groovyllä on mahdollista vähentää listat toisistaan, jolloin jäljelle jää eroavaisuudet. Tässä kohtaa tosin tiedetään ainoastaan eroavaisuuksia olevan, mutta ei tiedetä onko kenttiä poistunut vai tullut lisää. Tämän selvittämiseksi tein pienen vertailun (Kuva 9).

```
47  // Logic for determining if dataobjects gained or lost fields
48  if(difference.size() == 0){
49      println "No changes in DataObjects"
50  }
51  else {
52      println "New fields added to ${objectName}"
53      updated.each { item ->
54          if(!original.contains(item)) {
55              addedFields.add(item)
56          }
57      }
58      println addedFields
59
60      println "Removed fields from ${objectName}"
61      original.each {item ->
62          if(!updated.contains(item)) {
63              removedFields.add(item)
64          }
65      }
66      println removedFields
67  }
```

Kuva 9. Vertailu jolla selvitetään uudet ja poistuneet kentät.

Ohjelma katsoo ensin, onko eroavaisuuksien listan koko suurempi kuin 0. Jos listan koko on 0, silloin dataobjektissa ei kenttäkoodien suhteen ole tapahtunut muutosta. Jos taas listan koko ei ole 0, iteroidaan kenttäkoodit sisältävät listat läpi ja katsotaan sisältääkö vertailussa oleva lista vastaavat kenttäkoodit. Jos muutoksia löytyy, tallennetaan ne omiin listoihinsa.

Jos syntyneet listat tulostetaan ulos, saadaan alla olevan esimerkin kaltaista tulosta (Kuva 10).

```
New fields added to lomalista
[lomiltapaluu_pvm]
Removed fields from lomalista
[]

Process finished with exit code 0
```

```
New fields added to lomalista
[lomiltapaluu_pvm]
Removed fields from lomalista
[Lomarahat]

Process finished with exit code 0
```

Kuva 10. Kaksi esimerkkiä tulosteista.

Testatakseni vertailua ohitin tietokantahaun hetkeksi ja tein lukuisia erilaisia koodilistoja. Pyrin näillä simuloimaan dataobjektin päivitystä erilaisin lopputuloksin. Testaamani tilanteet jäivät kaikki talteen. Alkuperäisenä listana käytin aina samaa ns. vanhaa dataobjektia ja vertailin uusia listoja sitä vasten. Vertailulistoissa oli uusia kenttiä, poistuvia kenttiä sekä näiden sekoituksia.

4.3.5 Raportin luonti MongoDB-tietokantaan

Nyt kun vertailu toimii ja tulokset saadaan ulos järkevästi, on aika suunnitella raporttidokumentti MongoDB-tietokantaan. Loin raportteja varten oman tietokannan nimeltä migrationreports ja sinne kokoelman, johon raporttidokumentit voidaan tallentaa. MongoDB-dokumentin luonti on hyvin yksinkertaista (Kuva 11). Luodaan uusi tietokantaobjekti ja sen kentille annetaan nimi sekä arvo. Lopulta luotu objekti tallennetaan insert-metodilla.

```
// Adding document containing changes into the mongodb
def now = new Date()

DBObject document = new BasicDBObject("Migration Date", now)
        .append("Task", version)
        .append("DataObjekti", objectName)
        .append("New Fields", addedFields)
        .append("Removed Fields", removedFields)
recordCollection.insert(document)
```

Kuva 11. MongoDB-dokumentin luontia.

Raporttidokumenttiin lisäsin kentät skriptin ajopäivämäärälle, Jira-tehtävänumerolle, dataobjektin nimelle, uusille kentille sekä poistuville kentille. Uudet kentät sekä poistuvat kentät ovat rakenteeltaan listoja, mutta ne voidaan lisätä yhtälailla append-metodilla kuin mitkä tahansa muut kentät. Dokumentin rakentava funktio osaa ottaa datatyypit huomioon ja lisätä ne oikeanlaisena tietokantaan.

Kuvassa 12 on kuvattuna miltä raporttidokumentti näyttää, kun skriptin on ajanut, ajossa ollut yksi uusi kenttä ja kolme poistuvaa kenttää.

(1) {_id : 5d287bd90be5f42045b298b1}	{ 6 fields }
_id	5d287bd90be5f42045b298b1
Migration Date	2019-07-12T12:23:53.433Z
Task	ABC-1234
DataObjekti	JokuObjekti
New Fields	[1 elements]
0	UusiKohde
Removed Fields	[3 elements]
0	PoistuvaKohde1
1	PoistuvaKohde2
2	PoistuvaKohde3

Kuva 12. Esimerkki muutosten raportoinnista MongoDB-tietokannassa.

4.3.6 Koodin ajon turvaaminen ja loppusilaukset

Mikäli skriptin ajon aikana ilmenee ongelmia, esimerkiksi annetaan väärä tietokantanimi tai dataobjektin nimessä on kirjoitusvirhe, aiheutuu NullPointerException. Tämä yleisesti tarkoittaa sitä, että ohjelmakoodi yrittää suorittaa osiota, jota ei ole olemassa ja lopputuloksena on ohjelman kaatuminen. Tämän estämiseksi päätin käyttää try/catch-ominaisuutta. Tämä tarkoittaa sitä, että ajettava ohjelmakoodi laitetaan try-koodiblokin sisään ja jos koodin ajossa ilmenee vakava virhe, catch-koodiblokki sieppaa virheen talteen. Tällä pyritään estämään virheiden eskalointi. Poikkeuksen aiheuttama virhe on usein niin vakava, että koko ohjelma kaatuu tai aiheuttaa vakavia ongelmia myöhemmässä vaiheessa ohjelman ajoa. Jos taas poikkeus saadaan siepattua käyttämällä try/catch-ominaisuutta, voidaan ohjelmakoodi joko ajaa alas hallitusti tai virhetilanteelle voidaan tehdä jotain ja ohjelmakoodin ajoa voidaan mahdollisesti jatkaa.

Lopuksi lähdin luomaan rakennetta skriptin ajolle. Ajatuksena oli, että käyttäjä määrittelee itse vertailun kohteena olevan dataobjektin sekä tietokannan, jossa dataobjektin tiedot sijaitsevat. Tätä varten, mikäli käyttäjä ei skriptin käynnistyksen yhteydessä kirjoita komentoriviargumenteiksi Jira-tehtävänumeroa, dataobjektin nimeä tai tietokannan nimeä, loin mahdollisuuden antaa kyseiset tiedot skriptin käynnistyttyä. Mikäli nämä tiedot on annettu, niitä ei erikseen enää kysytä.

Tässä kohtaa ensimmäinen versio oli valmis. Työn toimeksiantajan palaute skriptin ensimmäisestä versiosta oli positiivista ja kannustavaa. Seuraavaa versiota varten sain paljon kehitysideoita ja käyn niitä läpi tulevassa kappaleessa.

4.4 Toinen versio

Ensimmäisen version hyvän palautteen avulla päätin tässä toisessa versiossa kohdentaa vertailun kokonaan toiseen kokoelmaan. Ensimmäisessä versiossa otin vertailtavan datan kokoelmasta, joka ei välttämättä ole kaikista parhain lopputavoitetta ajatellen. Tässä parannellussa versiossa vertailtava data tulee kokoelmasta, jossa dataobjektit on koottu ohjelmakohtaisesti. Tätä kokoelmaa käyttäen pystyn kerralla tarkistamaan kaikki dataobjektit, kun edellisessä versiossa skripti kohdistui yhteen dataobjektiin kerrallaan. Tämä lähestymistapa myös vie skriptiä kohti lopullista tavoitetta, joka on saada se osaksi jatkuvan integraation putkea.

Tässä luvussa hyödynnetään edellisen version ohjelmakoodia paljon eikä täten samoja asioita kannata käydä läpi.

4.4.1 Suunnittelu

Pystyin suurilta osin hyödyntämään edellisen version toiminnallisuutta, joten tämän seuraavan version suunnittelu koskee ainoastaan muuttuvia osia. Lähdin toteuttamaan seuraavaa versiota alla olevan luetelman mukaan.

- Otetaan talteen kaikki dokumentit templaattikokoelmasta. Jokainen dokumentti on yksi dataobjekteista.
- Iteroidaan jokainen dataobjekti läpi, luodaan muutoksista Map (Java Dictionary).
  - Uuden Mapin jokaisen elementin avain on dataobjektin nimi.
  - Avainta vastaava arvo on kokoelma muutoksista, joita dataobjektille on tullut.
    - Tallennetaan päivämäärä.
    - Lista uusista/poistuneista kentistä.
    - Lista muuttuneista kenttien nimiarvoista.
    - Lista mahdollisista tyhjistä arvoista (teksteissä).

Lopputavoitteena siis on Map-elementti, jonka koko on dataobjektien lukumäärä. Avaimina toimii dataobjektien nimet ja näiden arvoina listarakenteet, jotka sisältävät mahdolliset muutokset.

4.4.2 Tietojen haku ja dataobjektien tarkastaminen

Pohjana käytin ensimmäisen version koodia. Tietokantahakua muutin siten, että poistin mahdollisuuden käyttäjän antamille syötteille, joita olivat Jira-tehtävänumero, dataobjektin nimi sekä tietokannan nimi. Tietokantahakuun laitoin koko templaattikokoelman ja ohjasin sen lopulta listamuotoon. Tällä tavoin sain kokoelman jokaisen dokumentin omaksi elementikseen listarakenteeseen ja testaamisen vuoksi minulla on kokoelma alkuperäisistä dataobjekteista sekä muuttuneista dataobjekteista.

Dataobjektien haun jälkeen järjestän molemmat kokoelmalistat aakkosjärjestykseen niiden nimien suhteen. Tämän jälkeen otan alkuperäisten dataobjektien listalta dataobjektien järjestyneet nimet talteen erilliseen listaan, tämä lista toimii lopullisen tulosraportin avainlistana. Avaan tätä ajatusta myöhemmässä kohtaa.

Aikaisemmassa versiossa, jos dataobjektia ei löytynyt, tuli vain yksinkertainen virheviesti, joka pyysi tarkastamaan annetun dataobjektin nimen kirjoitusvirheiden varalta. Koska nyt prosessin on toimittava automaattisesti, tällaista ratkaisua ei voi tehdä. Käytin aiemmin aakkosjärjestykseen lajiteltuja listoja hyödyksi tarkistaessani, onko dataobjekteja kadonnut tai kenties tullut lisää. Tein tarkistuksen siten, että vertasin alkuperäisen dataobjektilistauksen kokoa päivitettyjen dataobjektien listan kokoon. Jos päivitettyjen dataobjektien lista on suurempi, tarkoittaa se silloin sitä, että uusia dataobjekteja on tulossa.

Tilanteesta riippuen koodi etenee kohtaan, jossa suoritetaan vertailu dataobjektien nimien välillä. Jos alkuperäisestä listauksesta löytyy dataobjektin nimi, jota ei enää ole päivitetyssä listauksessa, siirretään dataobjektin nimi kadonneiden dataobjektien listalle odottamaan tulosten vientiä Mongo-tietokantaan (Kuva 13). Sama prosessi uusille dataobjekteille. Samassa myös poistetaan tarkistuksessa kiinni jääneet dataobjektit lopullisesta muutosvertailusta, tällä pidetään vertailuun menevä data yhdenmukaisena, vertaillaan vain siis olemassaolevien dataobjektien mahdollisia muutoksia.

```
def tempList = []

// Check if there are new or missing DataObjects
if(originalDataObjectsAll.size() > updatedDataObjectsAll.size()) {
    // Lost DataObject
    collectedOriginalDataObjects.each { item ->
        if(!collectedUpdatedDataObjects.contains(item)) {
            listOfRemovedDO.add(item)
        }
    }
    // Remove lost DataObject from comparison
    collectedOriginalDataObjects.removeAll(listOfRemovedDO)
    tempList = originalDataObjectsAll - (originalDataObjectsAll - updatedDataObjectsAll)
    originalDataObjectsAll = tempList
}
else if(originalDataObjectsAll.size() < updatedDataObjectsAll.size()) {
    // Gained DataObject
    collectedUpdatedDataObjects.each { item ->
        if(!collectedOriginalDataObjects.contains(item)) {
            listOfNewDO.add(item)
        }
    }
    // Remove new DataObject from comparison
    collectedUpdatedDataObjects.removeAll(listOfNewDO)
    tempList = updatedDataObjectsAll - (updatedDataObjectsAll - originalDataObjectsAll)
    updatedDataObjectsAll = tempList
}
```

Kuva 13. Uusien ja kadonneiden dataobjektien tarkistus.

Tässä kohtaa keskustelin myös opinnäytetyöohjaajani kanssa siitä, onko tarvetta seurata dataobjekien kentän tekstielementtien muuttumista. Tämä päätettiin jättää nyt kokonaan pois. Ominaisuus on kuitenkin tulevaisuudessa kohtalaisen helppo lisätä takaisin, mikäli sille nähdään tarvetta.

4.4.3 Vertailu

Dataobjektien tarkistamisen jälkeen päästäännäiden sisälle muuttuneen tiedon vertailuun. Tämä koodiosio on hyvin samankaltainen kuin edellisessä versiossa. Koska kyseessä on nyt tilanne, jossa suoritetaan vertailu kaikille dataobjekteille, pitää ne kaikki iteroida läpi. Pistin siis edellisen version koodin ympärille for-silmukan, joka käy kaikki järjestyksessä läpi.

Tässä kohdin huomasin ongelman. Jostain syystä aivan kaikki dataobjektien muutokset eivät jääneet tarkistuksessa kiinni. Aluksi epäilin syyn johtuvan muutaman dataobjektin poikkeuksellisesta rakenteesta, mutta syy olikin muualla. Ongelman syyksi paljastui vertailulogiikassa käyttämäni metodit (Kuva 9) sekä identtiset kenttäkoodit. Aiemmin mainitsin, että kenttäkoodit ovat yksilöllisiä. Kävi ilmi, että kenttäkoodien on toki oltava yksilöllisiä, mutta ainoastaan sen kentän kyseisen ryhmän sisällä. Jossain toisessa ryhmässä saa olla kenttä samalla koodilla. Dataobjektien kaikki kysymykset ovat jaettu setteihin ja näiden sisällä vielä ryhmiin.

Metodit, joita käytin vertailussa eivät yksilöi identtisiä arvoja, vaan mikäli löytyy osuma, vaikuttaa se listassa oleviin kaikkiin vastaaviin arvoihin. Tällöin, jos dataobjektilla on esimerkiksi kenttäkoodi pvm ja lisään uuden pvm kentän toiseen kysymysryhmään, koodi ei tunnistanut sitä. Kirjoitin siis vertailulogiikan uudelleen (Kuva 14) ja nyt se toimii, kuten sen pitääkin.

```
def tempOrig = listOfCodesInDO_orig.collect()
def tempUpd = listOfCodesInDO_updated.collect()

// Comparison
listOfCodesInDO_orig.each { item ->
    if(tempUpd.contains(item)) {
        tempUpd.remove(item)
    }
}
results.put("Added Fields", tempUpd)

listOfCodesInDO_updated.each { item ->
    if(tempOrig.contains(item)) {
        tempOrig.remove(item)
    }
}
results.put("Lost Fields", tempOrig)

fullResults.put(collectedOriginalDataObjects[i], results)
}
```

Kuva 14. Uusi paranneltu vertailu.

Ylläolevasta kuvasta nähdään, että vertailun löydökset lisätään results-nimiseen datarakenteeseen. Vertailun pääsilmukassa jokaisella kierroksella alustetaan puhdas dataobjektikohtainen Map-rakenne results, jonne lisätään löytyneet muutokset. Rakenteen sisälle loin kentät uusille sekä poistuneille koodikentille (Kuva 15), nämä kentät ottavat sisäänsä listarakenteen. Joten vertailussa määritellään avaimena joko *Added Fields* tai *Lost Fields* ja muodostunut tulosten listaus lisätään tämän avaimen arvoksi.

```
89
90    def results = ["Added Fields":[], "Lost Fields":[]]
91
```

Kuva 15. Map-rakenne dataobjektin tuloksille.

Silmukan lopussa dataobjektikohtaiset tulokset lisätään lopulliseen Map-rakenteeseen *fullResults,* jossa avaimena toimii kappaleessa 4.4.2 mainitun dataobjektien nimilistan elementti (kts. kuva 16.). Avain määräytyy silmukan senhetkisen iteraattorin arvon mukaan ja lopullinen tulos on muotoa:

Dataobjektin nimi – [Added Fields:[koodikentän nimi], Lost Fields:[koodikentän nimi]]

4.4.4 Raportin luonti MongoDB-tietokantaan

Mongo-tietokantaan menevää raporttidokumenttia ei ollut tarpeen muuttaa suuremmin. Poistin edelliseen versioon liittyvät arvot ja lisäsin kohdat uusille ja poistuneille dataobjekteille (Kuva 17).

```
// Adding document containing changes into the mongodb
def now = new Date()

DBObject document = new BasicDBObject("Migration Date", now)
        .append("Changes per DataObjekti", fullResults)
        .append("New DataObjects", listOfNewDO)
        .append("Lost DataObjects", listOfRemovedDO)
recordCollection.insert(document)
```

Kuva 16. Päivitetty raporttidokumentti.

Kuvassa 18 on hieman muokattu raporttidokumentti ajosta. Korvasin oikeat dataobjektien nimet sekä kenttien nimet geneerisillä nimillä. Kuvasta voidaan nähdä, että ajosta syntyy dokumentti kokoelmaan. Dokumentista nähdään dataobjektikohtaisesti uudet ja poistuneet kentät sekä uudet että poistuneet dataobjektit. Kenttien ja dataobjektien nimet ovat value-sarakkeessa.

Key	Value
(12) {_id : 5d563a4ecac6f1f25de24076}	{ 5 fields }
_id	5d563a4ecac6f1f25de24076
Migration Date	2019-08-16T05:08:30.556Z
Changes per DataObjekti	{ 3 fields }
DataObjekti A	{ 2 fields }
Added Fields	[1 elements]
0	Uusi Kohde
Lost Fields	[0 elements]
DataObjekti B	{ 2 fields }
Added Fields	[1 elements]
0	UUSIOPTIO
Lost Fields	[0 elements]
DataObjekti C	{ 2 fields }
New DataObjects	[0 elements]
Lost DataObjects	[1 elements]
0	Poistunut DataObjekti

Kuva 17. Esimerkki raporttidokumentista Mongo-tietokannassa.

Tämän hetkinen versio skriptistä tekee tehokkaasti sen, mitä siltä odotetaankin. Seuraava etappi on viedä skripti Bamboo-palvelimelle. Skriptiin ei todennäköisesti jouduta tekemään kovinkaan suuria muutoksia. Seuraavaksi lähden tutkimaan, miten Bamboo toimii ja katson saanko skriptin sinne osaksi putkea.

4.5 Skriptin vieminen Bamboo-palvelimelle

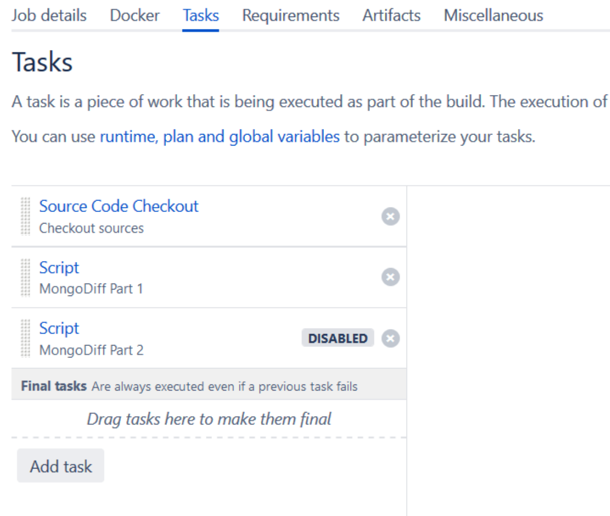
Viimeinen vaihe toimeksiannossa on viedä skripti osaksi Bamboo-palvelimen jatkuvan toimituksen putkea. Bamboo oli itselleni täysin vieras ympäristö, mutta sain kollegalta erittäin hyviä vinkkejä, joiden avulla pääsin sujuvasti alkuun. Tässä kappaleessa käyn läpi toimenpiteet, joilla sain skriptin vietyä Bamboo-palvelimelle.

4.5.1 Ympäristön luonti ja testiajot

Päätin, että jaan skriptini kahteen osaan. Ensimmäisen osan tehtävänä on hakea tietokannasta dataobjektien nykyinen tilanne ja tallentaa tämä tiedostoon odottamaan myöhempää käyttöä. Skriptin toisen osan on tarkoitus avata aiemmin luotu tiedosto sekä tämän jälkeen hakea tietokannasta päivitetty tilanne ja suorittaa vertailu. Syy miksi päätin tehdä tämän näin liittyy toimituksen prosessiin. Skriptin ensimmäinen osa ajettaisiin putken alkupäässä ja skriptin toinen osa putken loppupäässä, näiden välillä suoritetaan putken normaalit toimet.

Toimeksiantajan Bamboo-palvelimella oli valmiiksi olemassa toimitusprojekti sekalaisille toimenpiteille. Sain luvan tehdä kyseisen projektin sisään oman suunnitelman (Bamboossa Plan) nimeltä MongoDiffTestailua. Suunnitelmia voidaan kustomoida tekemään toimittamisen aikana useita erilaisia toimintoja ja tehtäviä. Tässä tilanteessa linkitin ensin suunnitelmaan sen GitLab-repositorion, jonne olin nyt kahtia jakautuneen skriptini tallentanut. Versionhallinnan yhdistäminen Bamboon suunnitelmaan on erinomainen idea, sillä koodimuutokset heijastuvat välittömästi oikeaan paikkaan.

Repositorion linkittämisen jälkeen loin suunnitelmalleni työtehtävän ja tälle toimintoja (Tasks). Ensimmäinen toiminto on hyvin oleellinen Repository Checkout, joka hakee linkitetyn repositorion sisällön, mikäli se havaitsee muutoksia. Checkout-toiminnon jälkeen loin kaksi script-toimintoa, nämä tukevat Bamboon sisäisiä skriptejä eivätkä liity toimeksiannon skriptiin varsinaisesti (Kuva 19).



Kuva 18. Bamboo-tehtävän toiminnot.

Ympäristö on nyt valmis testailuun. Lähdin kirjoittamaan ensimmäiseen Bamboon skriptitoimintoon testikoodia, jotta pystyin varmistamaan sen toiminnan. Kirjoitin yksinkertaisia shell-komentoja kuva 19 näkyvään MongoDiff Part 1 nimiseen skriptitoimintoon (Kuva 20).

Script body*

```
1  #!/bin/bash
2  set -e
3  . ${SDKMAN_DIR}/bin/sdkman-init.sh
4
5  groovy -v
6  ls oppari/src
7
8  groovy oppari/src/TestingScript.groovy
```

Kuva 19. Shell-komentoja testaamista varten.

Bamboon skripti-ikkunaan voi kirjoittaa koodia suoraan käyttäen Bash- tai PowerShell-tulkkia, mutta ajettavan tiedoston voi myös ilmoittaa erikseen kuten olen tehnyt rivillä 8. Kuvan 20 komennot tulostavat näytölle käytössä olevan Java- sekä Groovy-versiot, oppari/src -polussa olevat skriptit sekä ajaa TestingScript-nimisen tiedoston. Tiedoston sisällä ei ole muuta kuin yksi Groovy-komento, joka tulostaa sanan Hello. Alla olevasta kuvasta (Kuva 21) voimme nähdä ajon tulokset.

Summary Tests Commits Artifacts Logs Metadata Issues

Logs

The following logs have been generated by the jobs in this plan.

Job

Default Job Default Stage

```
16-Jul-2019 11:54:07 Picked up _JAVA_OPTIONS: -Dfile.encoding=utf-8 -Dsun.jnu.encoding=utf-8
16-Jul-2019 11:54:07 Groovy Version: 2.1.9 JVM: 1.8.0_201 Vendor: Oracle Corporation OS: Linux
16-Jul-2019 11:54:07 TestingScript.groovy
16-Jul-2019 11:54:07 mongo_diff.groovy
16-Jul-2019 11:54:07 mongo_diff2.groovy
16-Jul-2019 11:54:07 Picked up _JAVA_OPTIONS: -Dfile.encoding=utf-8 -Dsun.jnu.encoding=utf-8
16-Jul-2019 11:54:09 Hello
```

Kuva 20. Bamboo-testiajon tulokset.

Kaikki näyttää toimivan oikein, joten seuraavaksi tuodaan oikeat skriptit mukaan toimintaan.

4.5.2 Skriptin sovitus osaksi putkea

Koska olin jakanut skriptini kahteen osaan, minun piti löytää keino, jolla saan siirrettyä tietoa skriptistä toiseen. Kokeilin siirtämistä käyttäen yksinkertaista tekstitiedostoon tallentamista ja sen lukemista toisessa skriptissä. Tässä vaiheessa en ollut vielä ratkonut sitä, kuinka pääsen Bamboo-ajoissa kiinni oikeisiin tietokantoihin, joten minun piti soveltaa. Tein muualla tietokantahaun ja tallensin hakutulokset tekstitiedostoon ja tämä toimii eräänlaisena varatietokantana. Käyttäen tätä tiedostoa apuna luin sen sisällön ensimmäisessä skriptissä ja tallensin sen uuteen tekstitiedostoon dbOrig.txt.

Skriptin toisessa osassa luin tämän tiedoston muuttujaan, jonka jälkeen tulostin sen ensimmäisen elementin sisällön Bamboon loki-ikkunaan ajon yhteydessä ilman ongelmia.

Nyt, kun tiesin, että vastaava tiedonsiirto onnistuu, lähdin suunnittelemaan oikeaa siirtoprosessia. Tämä siksi, että tekstimuotoinen data ei sovellu lopputarkoitukseen. Päädyin kahteen vaihtoehtoon, joko tallentaisin tietokantahaun JSON-muodossa tai käyttäisin serialisointia datan pakkaamisessa. Päädyin käyttämään JSON:a. Sille löytyy erinomaiset metodit Groovystä ja se on muutenkin erittäin yleinen keino siirtää tietoa.

JSON-datan muodostus Groovyllä on yksinkertaista (Kuva 22) käyttäen jsonOutput-kirjastoa. Samalla, kun muodostan JSON-tiedoston, järjestän dataobjektit aakkosjärjestykseen.

```
def json = JsonOutput.toJson(originalDataObjectsAll.sort { it.code })
new File( pathname: "data.json").write(json)
```

Kuva 21. JSON-tiedoston luominen.

Ja JSON-datan lukeminen muuttujaan on yhtä yksinkertaista kuin sen luonti (Kuva 23).

```
def jsonSlurper = new JsonSlurper()
File incomingFile = new File( pathname: "oppari/src/data.json")
def fileContents = incomingFile.readLines()
def originalDataObjectsAll = jsonSlurper.parseText(fileContents)
```

Kuva 22. JSON-tiedoston lukeminen toisessa skriptissä.

Nyt kun minulla on tallessa alkuperäiset dataobjektit JSON-muodossa, otin tiedostosta kopion ja tein sinne muutamia muutoksia. Nämä muutokset simuloivat muuttuneita koodikenttiä eri dataobjekteissa. Skriptin varsinaiseen vertailulogiikkaan ei tarvitse tehdä mitään muutoksia, ainoat lisät liittyvät tallennetun tiedoston lukemiseen.

Kun olen saanut kaikki valmiiksi, päivitän koodin repositorioon ja sinne siirtyvät myös luodut JSON-tiedostot. Tämän jälkeen käynnistän ajon, Bamboo hakee muutokset ja suorittaa skriptit, ajo on onnistunut ja loki-ikkunaan tulostuu vertailussa kiinni jääneet muuttuneet kentät.

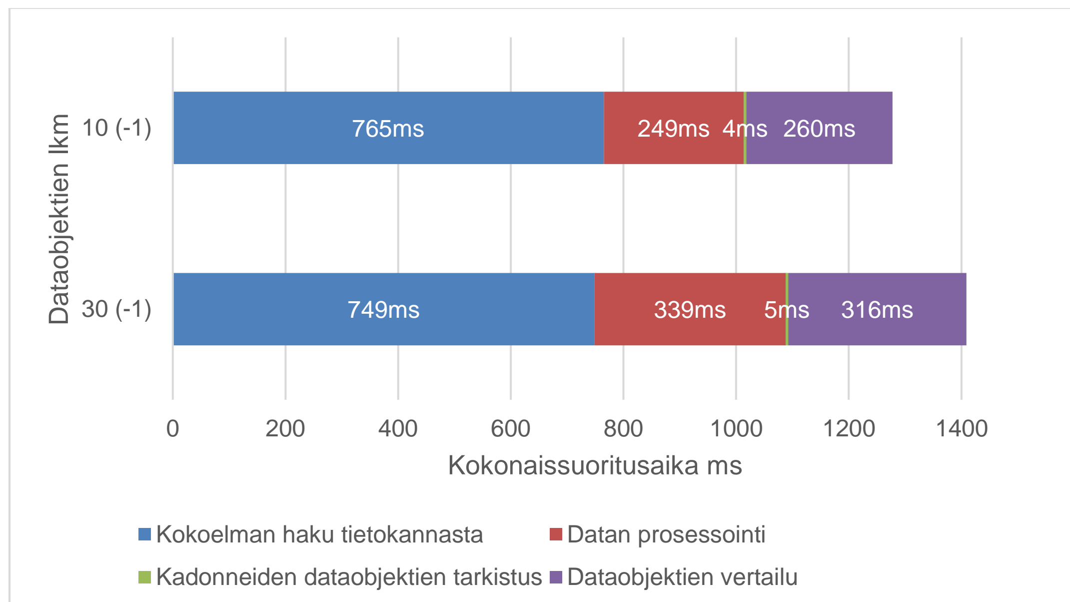
Koska Bamboo-ajossa simuloin tietokantoja JSON-tiedostoilla, haluan vielä varmistaa skriptien toiminnan lokaalissa ajossa IntelliJ IDEA:n avulla. Käynnistän ensimmäisen skriptin ja se suoriutuu oikein. Se hakee tietokannasta dataobjektien nykytilanteen ja tallentaa sen JSON-muodossa. Käynnistän toisen skriptin tämän jälkeen ja se suoriutuu myös ongelmitta. Se lukee JSON-tiedoston, hakee tietokannasta päivitetyn tilanteen ja suorittaa vertailun onnistuneesti, saan saman tuloksen kuin aikaisemmassa Bamboo-ajossa. Tämä ajo on hyvin lähellä todellista tilannetta, ajoympäristönä vain IntelliJ IDEA Bamboon sijaan.

5 TULOKSET

5.1 Skriptin toiminta

Bamboo-ajon tulosteiden lisäksi halusin vertailla myös skriptin suorituskykyä eli sitä, miten sen eri osa-alueet toimivat eri määrillä dataobjekteja. Skriptin rakennetta tutkiessa päädyin siihen, että tämän asymptoottinen suoritusaika on $O(n^2)$ n:n ollessa dataobjektien määrä. Toimeksiantajan ohjelmissa dataobjekteja on vaihtelevasti 10 ja 30 välillä, vaikka skriptin suoritusaika teoreettisesti kasvaa lopulta hyvinkin jyrkästi dataobjektien määrän kasvaessa, ei vähäisen dataobjektien määrän vuoksi tästä tule koitumaan ongelmia skriptin ajon aikana.

Halusin myös nähdä, miten koko skripti ajautuu ja käyttää kokonaisaikansa, joten tein aikapisteitä jokaisen erillisen toiminnon väliin. Skriptin alussa ladataan GMongo-kirjasto, mutta tämän ollessa vain kertalataus en lisännyt sitä suoritusajan mittaukseen.



Kaavio 1. Skriptin toiminta purettuna osiin. Suoritusajat vs. dataobjektien määrä.

Kaaviosta 1 nähdään skriptin eri osa-alueiden suoritusajat. Otin vertailuun 10 ja 30 dataobjektia. Halusin myös aktivoida puuttuvien dataobjektien tarkistuksen, joten poistin yhdet päivitettyjen dataobjektien kokoelmista kummastakin, kaavion -1 tarkoittaa siis tätä. Ajoin skriptin 30 kertaa erillisinä käynnistyksinä ja otin jokaisella kierroksella ajat

ylös tietokantahausta, datan prosessoinnista, kadonneiden dataobjektien tarkistuksesta sekä itse vertailusta. Laskin näistä keskiarvot, joita käytin kaavion muodostamiseen. Nähtävillä on selkeästi, miten kokoelmien haku tietokannasta kestää lähes saman ajan 10 sekä 30 dataobjektilla. Sen sijaan datan prosessoinnissa alkaa jo näkymään, että 30 dataobjektilla aikaa kuluu hivenen kauemmin, sama vertailussa. Tämä on toki täysin odotettavissa, sillä kasvava dataobjektien määrä nostaa iteraatioiden ja operaatioiden määrää.

Normaalisti dataobjekteja ohjelmassa on 10 ja 30 välillä, joten suoritusaika vaihtelee 1300 – 1400 ms:n välillä. Huomioitavaa myös on se, että skriptin ollessa osa toimitusprosessia, ei nykyinen suoritusaika ole mitään, sillä varsinainen toimitusprosessi kestää rekisteristä riippuen 5 min – 15 min.

5.2 Skriptien ajo Bamboo-palvelimella

Luvun 4.5.2 lopussa suoritetun Bamboo-ajon tulos antaa loki-ikkunaan seuraavanlaisen tulosteen (Kuva 24). Muokkasin muutamien kenttäkoodien nimiä, jotka ilmenee kuvan mukaisena tulosteena. Lisätyissä kentissä näkyy kentän muokattu nimi ja poistuneissa kentissä vanha nimi.

#29 was successful – Manual run by Toni Huovinen

Summary Tests Commits Artifacts Logs Metadata Issues

Logs

The following logs have been generated by the jobs in this plan.

Job

Default Job Default Stage

```
18-Jul-2019 10:35:31 Picked up _JAVA_OPTIONS: -Dfile.encoding=utf-8 -Dsun.jnu.encoding=utf-8
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[ei_MUOKKAUS], Lost Fields:[ei]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[lisatietojaMUOKKAUS], Lost Fields:[lisatietoja]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
18-Jul-2019 10:35:37  - [Added Fields:[], Lost Fields:[]]
```

Kuva 23. Bamboo-ajon tuloste vertailusta.

Vastaavasti, jos dataobjektin jokin kenttä olisi poistunut kokonaan, näkyisi se ainoastaan poistuneiden kenttien kohdalla. Sama ajatus tietenkin myös kokonaan uusilla kentillä. Normaalisti kuvan 24 mukaista tulostetta ei välttämättä laitettaisi näkyviin loki-ikkunaan, eikä muuallekaan. Tuon tilalle laitetaan lopulta ilmoitus siitä, että dataobjektien vertailu on suoritettu onnistuneesti tai virheiden sattuessa ilmoitus on sen mukainen. Oikeastaan virheiden sattuessa halutaan, että koko toimitus kaatuu. Mongo-tietokannan muutosten automaattisesta seurannasta ei ole mitään hyötyä, jos prosessin annettaisiin mennä loppuun asti, eikä seuranta ole pysynyt perässä.

Lopullinen tulos tallennetaan ohjelman omalle tietokannalle sopivaan kokoelmaan, josta sen voi tarvittaessa aina hakea. Tällaisesta raporttidokumentista on esimerkkinä kuva 18.

Valitettavasti en onnistunut saamaan luomiani skriptejä haluttuun toimituksen putkeen mukaan. Tämä olisi onnistuakseen vaatinut tietynlaisia tietokantayhteyksiä osaksi skriptejä ja kesälomien vuoksi asiaan liittyvät oleelliset henkilöt olivat poissa tämän työn käytännön osuutta tehdessäni. Vaaditut tiedot olivat luonteeltaan sellaisia, etten niitä itse pystynyt selvittämään. Pystyin kuitenkin simuloimaan toimintaa Bamboossa ja IntelliJ IDEA:n kautta, kuten luvun 4.5.2 lopussa totesin.

6 YHTEENVETO

Yksi tämän työn taustatavoitteista oli selvittää, voiko Bamboo-ympäristöön luoda toiminnallisuutta, jolla Mongo-tietokantojen vertailun voi toteuttaa. Tutkimuksen tuloksena voidaan sanoa, että vertailun voi toteuttaa. Tässä opinnäytetyössä käytiin yksityskohtaisesti läpi dataobjektitemplaattien vertailun suorittavan skriptiparin ohjelmointi. Vertailu kohdistuu erityisesti dataobjektin koodikenttiin, jotka ovat tallennettavan datan kannalta tärkeät elementit. Vertailussa huomataan, jos joku koodikenttä ei välttämättä ole poistunut vaan sen arvoa on muokattu, mutta myös kokonaan poistuvat sekä uudet kentät on huomioitu. Vertailu myös tarkistaa mahdollisesti poistuneet tai uudet dataobjektit. Vertailun jälkeen tallennetaan löydökset Mongo-tietokantaan selkeään raporttikokoelmaan. Skriptien ajaminen myös Bamboo-ympäristössä toimii, mutta ne vaativat viralliset tietokantayhteydet ennen käyttöönottoa.

Tällä hetkellä toimeksiantajalla on dataobjektien rakenteen muodostamisessa käytössä kaksi omakehitteistä toisistaan hivenen poikkeavaa työkalua. Tässä opinnäytetyössä tehdyt skriptit on pääasiallisesti suunnattu yrityksen ohjelmille, jotka käyttävät työkaluista ensimmäistä. Mielenkiinnosta testasin skriptien toimintaa myös jälkimmäisellä työkalulla muodostettuihin dataobjekteihin ja muutosten vertailu näytti hyvältä.

Nähtäväksi jää, onko tästä opinnäytetyöstä työn toimeksiantajalle hyötyä. Ainakin osia koodista voidaan käyttää jatkokehitykseen ja työssä käytetyistä ideoista on varmasti hyötyä. Työn toimeksiantajalta tuli kannustavaa palautetta tehdystä työstä ja toivetta jatkokehitykselle. Itselleni työ oli mieluisa ja sopivan haastava. Tämä opinnäytetyö kehitti ohjelmointitaitojani sekä ongelmanratkontakykyä.

6.1 Jatkokehitysajatuksia

Skripteistä siis jäi puuttumaan viralliset tietokantayhteydet. Suunnittelin skriptien toimivan siten, että ensimmäinen hakee tietokannasta senhetkisen tilanteen dataobjekteista ja tallentaa sen JSON-muodossa Bamboo-palvelimelle. Kun tulee toisen skriptin vuoro toimia, tämä lukee aiemmin tallennetun JSON-tiedoston ja hakee tietokannasta talteen

uudet mahdollisesti päivittyneen tilanteen. Molemmissa tapauksissa tarvitaan rekisterikohtainen tieto siitä, mikä on tietokannan osoite, käyttäjätunnus ja salasana. Nämä tiedot tulevat mahdollisesti Bamboossa joistain ulkopuolisista muuttujista tai jonkin toisen skriptin ajon seurauksena. Etsinnöistä huolimatta en pystynyt tätä tietoa löytämään.

Tällä hetkellä skriptin vertailu tarkistaa dataobjektin kenttäkoodeja ja niiden muutoksia. Jos tulee uusi kenttä, sille annetaan kenttäkoodi ja tämä jää silloin vertailussa kiinni. Vastaavasti jos kenttä poistuu, jää siitä myös tieto. Myös itse kenttäkoodin mahdollinen muutos näkyy, kuten kuvassa 24 on esitettynä. Toimeksiannon alussa oli mainintaa kenttäkoodien tekstielementtien seurannasta, mutta tästä kuitenkin päätettiin luopua projektin selkeyttämisen vuoksi. Näiden tekstielementtien seuraaminen voisi olla yksi hyvä jatkokehittämisen kohde. On nimittäin hyvin yleistä, että jonkin kentän otsikko muuttuu.

Yksi tärkeimmistä lopputavoitteista oli luoda raporttidokumentti Mongo-tietokantaan. Tästä dokumentista kävisi ilmi onko mitään muutoksia tapahtunut versiopäivityksen yhteydessä. Onnistuin tässä kohtalaisen hyvin, mutta toki dokumentin rakenteessa voi olla aina parannettavaa. Tätä rakennetta on hyvä miettiä lopulta, että kattaako se tarpeeksi. Tällä hetkellä raporttiin tulee listaus jokaisesta dataobjektista ja jokaisen dataobjektin sisällä on paikat uusille sekä poistuville kentille. Myös uudet sekä poistuvat dataobjektit tallennetaan samaan raporttiin.

LÄHTEET

[1] SDLC – Software Development Life Cycle Overview [www-sivu] Saatavilla: https://www.tutorialspoint.com/sdlc/sdlc_overview.htm [Viitattu 8.6.2019]

[2] What is SDLC [www-sivu] Saatavilla: https://stackify.com/what-is-sdlc/ [Viitattu 8.6.2019]

[3] Compare MongoDB Collections, Studio3T [www-sivu] Saatavilla: https://studio3t.com/knowledge-base/articles/compare-mongodb-collections/ [Viitattu 21.6.2019]

[4] Wikipedia, IntelliJ IDEA [www-sivu] Saatavilla: https://en.wikipedia.org/wiki/IntelliJ_IDEA [Viitattu 8.6.2019]

[5] Java [www-sivu] Saatavilla: https://www.tutorialspoint.com/java/java_overview.htm [Viitattu 8.6.2019]

[6] Subraniam, V. *Programming Groovy 2*. Pragmatic Bookshelf 1st edition julkaistu 2013. 370 s. ISBN-13 978-1937785307 (painettu).

[7] Dan Vega. The Complete Apache Groovy Developer Course – Lecture 2. Saatavilla: https://www.udemy.com/tutorial/apache-groovy/what-is-groovy/ [Viitattu 10.6.2019]

[8] Continuous Integration [www-sivu] Saatavilla: https://www.thoughtworks.com/continuous-integration [Viitattu 10.6.2019]

[9] MongoDB ObjectId [www-sivu] Saatavilla: https://www.tutorialspoint.com/mongodb/mongodb_objectid.htm [Viitattu 10.6.2019]

[10] MongoDB [www-sivu] Saatavilla: https://www.tutorialspoint.com/mongodb/mongodb_overview.htm [Viitattu 10.6.2019]

[11] Mongo vs MySQL [www-sivu] Saatavilla: https://www.simform.com/mongodb-vs-mysql-databases/ [Viitattu 7.7.2019]

[12] What is MySQL, MySQL Documentation [www-sivu] Saatavilla: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html [Viitattu 23.6.2019]

[13] Nested Iteration [www-sivu, käyttäjä rboy, viesti kirjoitettu 28.7.2018] Saatavilla: https://stackoverflow.com/questions/51488434/how-to-iterate-through-all-values-of-a-nested-map-in-groovy [Viitattu 6.7.2019]