TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HARRI ESKOLA

OHJELMISTOSTANDARDIN KONELUETTAVA RAJAPINTA JA MUKAUTUVA WEB-KÄYTTÖLIITTYMÄ

Diplomityö

Tarkastajat:
professori Hannu Jaakkola
ja yliopistonlehtori Timo Mäkinen
Tarkastajat ja aihe hyväksytty
Talouden ja rakentamisen tiedekun-
taneuvoston kokouksessa 9.4.2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
ESKOLA, HARRI: Ohjelmistostandardin koneluettava rajapinta ja mukautuva web-käyttöliittymä
Diplomityö, 64 sivua, 26 liitesivua
Joulukuu 2014
Pääaine: Ohjelmistotekniikka
Tarkastajat: professori Hannu Jaakkola ja yliopistonlehtori Timo Mäkinen
Rahoittaja: TTY Pori, AVARAS-hanke
Avainsanat: koneluettava rajapinta, mukautuva käyttöliittymä, ohjelmistostandardi

Avoimella tiedolla tarkoitetaan digitaalista tietoa, joka on vapaasti saatavissa. Avoimien tietovarantojen hyödyntäminen on nouseva trendi maailmalla. Diplomityön tavoitteena oli toteuttaa elektroninen prosessiopas, joka havainnollistaa miten avoimia tietovarantoja hyödynnetään teknisesti.

Tässä tutkimuksessa keskityttiin vapaasti saatavilla oleviin JavaScript-pohjaisiin webtekniikoihin. Tutkimuksen tuloksena kehitettiin elektroninen prosessiopas, joka perustui pienten ohjelmistoyksiköiden prosessistandardiin käyttämällä moderneja webkehitystyökaluja. Lisäksi esiteltiin keskeisimmät tutkimuksessa käytetyt tekniikat.

Tutkimuksessa analysoitiin ja jäsennettiin pienten ohjelmistoyksiköiden prosessistandardi koneluettavaan muotoon JSON-dokumenteiksi, jotka talletettiin dokumenttitietokantaan. Tietokannalle toteutettiin koneluettava rajapinta noudattamalla REST-arkkitehtuurityyliä, jolla mahdollistetaan JSON-muotoisen datan noutaminen tietokannasta. Lopuksi toteutettiin JSON-muotoiseen dataan mukautuva web-käyttöliittymä elektroniselle prosessioppaalle, mikä noutaa oppaaseen liittyvää tietoa koneluettavan rajapinnan kautta JSON-dokumentteina.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
ESKOLA, HARRI: Machine and Human Readable Web Interfaces for a Software Process Standard
Master of Science Thesis, 64 pages, 26 Appendix pages
December 2014
Major: Software Engineering
Examiners: Professor Hannu Jaakkola and university lecturer Timo Mäkinen
Financing: TUT Pori, AVARAS project
Keywords: adaptable user interface, machine readable interface, software process standard

Open data refers to digital information which is freely available. The utilization of open data is a rising trend around the world. The objective of this study was to implement an electronic process guide which should provide information of how open data can be utilized.

This study concentrates on JavaScript based web technologies which are freely available. As the result of the study an electronic process guide was implemented. It is based on software process standard intended for very small entities using modern web development tools. In addition all technologies used on the study were presented.

Over the course of this study, software process standard intended for very small entities was analyzed and restructured into a machine readable JSON format. The JSON documents were then stored into a document-oriented database. A machine readable interface, which allows access to the JSON documents in the database, was implemented for the database following the REST architectural style. Finally a web user interface, which is capable of adapting to the JSON data, was implemented for the electronic process guide. The web application retrieves the software process related JSON formatted data through the machine readable interface.

ALKUSANAT

Olen tehnyt diplomityöni tutkimusapulaisena Tampereen teknillisen yliopiston Porin yksikössä. Työ syntyi osana AVARAS-hanketta.

Haluan kiittää professori Hannu Jaakkolaa, yliopistonlehtori Timo Mäkistä ja tutkimuspäällikkö Jari Soinia mahdollisuudesta tämän tutkimuksen tekemiseen. Haluan myös kiittää Jari Korpelaa, Petri Rantasta ja Pekka Sillbergiä työn tekemiseen liittyvistä vinkeistä, sekä koko TTY Porin yksikön henkilökuntaa miellyttävästä työilmapiiristä. Lisäksi haluan vielä erityisesti kiittää Timo Mäkistä työn ohjauksesta, neuvoista ja rakentavista kommenteista.

Porissa 19.12.2014

Harri Eskola
Antinkatu 25 B 30
28100 Pori

SISÄLLYS

TERMIT JA NIIDEN MÄÄRITELMÄT

Ajax	Asynchronous JavaScript And XML. Joukko web-sovelluskehityksen tekniikoita, joilla voidaan tehdä web-sivusta vuorovaikutteisempi.
API	Application Programming Interface. Ohjelmointirajapinta, jolla määritellään kuinka eri ohjelmat voivat vaihtaa tietoja keskenään.
AVARAS	EAKR-rahoitteinen hanke, joka keskittyy avointen tietovarantojen hyödyntämistä tukevan tiedon keräämiseen ja tuottamiseen Satakuntalaisen elinkeinoelämän tarpeiden pohjalta.
CRUD	Create, Read, Update and Delete. Luo, lue, päivitä ja poista ovat tiedon varastoinnin neljä perustoimintoa.
CSS	Cascading Style Sheets. Tyylikuvauskieli, jolla voidaan määrittää ohjeet WWW-sivun ulkoasulle.
Dojo Toolkit	Avoimeen lähdekoodiin perustuva modulaarinen JavaScript-kirjasto.
DOM	Document Object Model. Dokumenttioliomalli on HTML-dokumenttien sisällönmuokkauksen ohjelmointirajapinta.
EAKR	Euroopan aluekehitysrahasto.
ECMA	Kansainvälinen ja yksityinen voittoa tavoittelematon standardien organisaatio, joka on nykyään Ecma International.
Failover-proseduuri	Proseduuri, jolla järjestelmä siirtää kontrollin automaattisesti duplikaattijärjestelmälle huomattuaan vian tai häiriön.
HTML	Hypertext Markup Language. Web-sivujen totetuttamisen merkintäkieli.
HTTP	Hypertext Transfer Protocol. Selaimen ja palvelimen tiedonsiirtoon käytettävä protokolla.

I/O	Input/Output. Tiedon siirtämistä tai signaloimista eri komponenttien välillä.
IEC	International Electrotechnical Commission. Sähkö-, elektronisia- ja liittyviä teknologioita käsittelevä standardoimisorganisaatio.
IETF	Internet Engineering Task Force. Kehittää ja edistää vapaaehtoisia Internet-standardeja.
i18n	Prosessi, jolla kansainvälistetään sovellus toimimaan eri kielillä.
ISO	International Organization for Standardization. Kansainvälinen standardeja asettava elin.
JSON	JavaScript Object Notation. Kevyt tiedonsiirtoformaatti.
MongoDB	NoSQL-tietokannaksi luokiteltava dokumenttitietokanta.
NPM	Pakettienhallintatyökalu Node.js-alustalle, jota on aiemmin kutsuttu Node Packaged Modules -nimellä.
PM	Project Management. Projektinhallintaprosessi pienten ohjelmistoyksiköiden ohjelmistostandardissa.
RAM	Random Access Memory. Keskusmuisti eli työmuisti, johon latautuvat käyttöjärjestelmän ohjelmat.
RFC	Request for Comments. IETF-organisaation Internetiä koskeva standardiperhe.
REST	REpresentational State Transfer. Arkkitehtuurityyli ohjelmointirajapintojen toteuttamiselle.
SI	Software Implementation. Ohjelmistontoteutusprosessi pienten ohjelmistoyksiköiden ohjelmistostandardissa.
SOAP	On aiemmin ollut akronyymi sanoista Simple Object Access Protocol. Spesifikaatio jäsennellyn tiedon vaihtamiselle web-palveluissa.

URI	Uniform Resource Identifier. Merkkijono, jolla kerrotaan tiedon yksikäsitteinen nimi tai paikka.
VSE	Very Small Entity. Pieni ohjelmistoyksikkö, jossa on enintään 25 työntekijää.
Widget	Pienoisohjelma. Sillä on yleensä näkyvä olomuoto selainikkunassa ja se niputtaa loogisesti yhteen yhdeksi olioksi tai komponentiksi HTML-, CSS-, JavaScript-, sekä staattiset resurssit.
WSDL	Web Service Description Language. XML-pohjainen rajapintojenmäärittelykieli.
XML	Extensible Markup Language. On merkkauskieli, joka määrittelee säännöt dokumenttien koodaamiselle formaatilla, joka on sekä ihmisten että koneluettava.

1 JOHDANTO

Nykypäivän web-sovellukselle on paljon haasteita ja vaatimuksia. Sovelluksen täytyy muiden muassa olla modernin näköinen, sen pitää latautua nopeasti ja yhdenmukaisella tavalla jokaisella latauskerralla, sen pitää olla kevyt eikä vaatia paljoa resursseja, sen pitää toimia usealla eri alustalla, sekä web-sovelluksen kehittämisen täytyy olla nopeaa ja vaivatonta. Lisäksi ohjelmistonkehitykseen liittyvät prosessit tulee olla kunnossa.

Tutkimuksessa rakennetaan elektroninen prosessiopas, joka perustuu pienten ohjelmistoyksiköiden prosessistandardiin. Oppaan toteutukseen valitaan avoimien tietovarantojen yhteydessä tänä päivänä käytettäviä teknologioita. Valituilla teknologioilla voidaan rakentaa valmiista komponenteista helposti ja vaivattomasti moderneja käyttöliittymiä, esimerkiksi avoimien tietovarantojen yhteydessä usein käytettyjä rajapintoja on yksinkertainen toteuttaa, sekä kaikki ohjelmistokehitys tapahtuu samalla kielellä.

Tutkimuksessa käytetyt teknologiat on tarkoitus esitellä laajemmin ja toteutetun elektronisen prosessioppaan on tarkoitus havainnollistaa, miten avoimia tietovarantoja voidaan hyödyntää.

Tässä luvussa esitellään tutkimuksen taustaa, sen ongelma ja rajaukset, tavoitteet ja menetelmät, sekä tutkimuksen rakenne esitellään viimeisenä.

Tausta

Diplomityö on toteutettu osana Tampereen teknillisen yliopiston Porin yksikön AVARAS-tutkimushanketta, jossa olin mukana tutkimusapulaisena. EAKR-rahoitteinen AVARAS -hanke (1.8.2013 – 31.12.2014) keskittyy avointen tietovarantojen hyödyntämistä tukevan tiedon keräämiseen ja tuottamiseen Satakuntalaisen elinkeinoelämän tarpeiden pohjalta. Hankkeen toteuttaa TTY Porin laitos Satakuntalaisten PK-yritysten ohjaamana. Yksi hankkeen päätavoitteista on selvittää miten avoimia tietovarantoja hyödynnetään teknisesti.

Ongelma ja rajaukset

On olemassa erilaisia ”paperimuotoisia” prosessiaineistoja (esimerkiksi standardeja kuten ISO/IEC 15504-5), näiden ongelmana on käytettävyys, niiden lukeminen voi olla hankalaa ja se vaatii vaivannäköä opetella lukemaan standardeja.

On myös ohjelmistoapuvälineitä, joilla voidaan laatia elektronisia prosessioppaita (esim. Eclipse Process Framework Composer), näiden ongelmana on taustalla oleva prosessimetamalli (prosessimallin malli), joka ei välttämättä vastaa aineiston metamallia. Toisena ongelmana on se, että tiedot joudutaan syöttämään lomakkeiden kautta.

Tavoitteet ja menetelmä

Tavoitteena olisi edellä mainittujen ongelmien lieventäminen siten että prosessiaineisto saataisiin JSON-muotoisena pakettina ja elektronisen prosessioppaan käyttöliittymä mukautuisi aineiston rakenteen mukaan. Tavoitteena olisi myös esitellä sovelluksen rakentamiseen liittyvät tekniikat hieman laajemmin. Sovellus rakennetaan noudattaen standardin ISO/IEC TR 29110-5-1-1:2012(E) määrittelemää prosessimallia.

Rakenne

Luvussa 2 esitellään pienten ohjelmistoyksiköiden prosessistandardi (ISO/IEC TR 29110-5-1-1:2012(E)), jolla on tässä työssä kaksijakoinen rooli. Standardia käytetään toteutetun elektronisen prosessioppaan testidatana sekä itse toteutuksen osalta seurataan standardin tarjoamaa prosessimallia ohjelmistontoteutusprosessin osalta.

Luvussa 3 esitellään tutkimuksen tekniikat tiedonhallintajärjestelmän sekä palvelin- ja selainpäissä käytettyjen tekniikoiden osalta. Tiedonhallintajärjestelmäosiossa esitellään JavaScript Object Notation -tietoformaatti (JSON) sekä MongoDB-dokumenttitietokanta. Palvelinpäänosiossa esitellään Node.js-palvelinalusta ja siihen ladattavia kolmannen osapuolen moduuleja sekä Representational State Transfer -arkkitehtuurityylin (REST) ominaisuuksia. Selainpäänosiossa esitellään Dojo-työkalupakin sekä JavaScript-ohjelmointikielen ominaisuuksia.

Luvussa 4 esitellään luvun 3 tekniikoita hyödyntävä sovellus. Luvun 4 ensimmäisessä aliluvussa esitellään toteutetun sovelluksen käyttötapaukset ja arkkitehtuuri. Seuraavaksi esitellään tietokannan osuus sovelluksessa eli miten tietokantaan talletettu data on luotu. Koneluettavan rajapinnan osiossa selvitetään Node.js -web-palvelinalustalle rakennetun REST-ominaisuuksia omaavan rajapintasovelluksen toimintaa. Mukautuvan käyttöliittymän osiossa selvitetään Dojo Toolkitilla rakennetun käyttäjälle näkyvän asiakassovelluksen toimintaa.

Luvussa 5 vedetään yhteen ja kerrataan tutkimuksen tulokset ja huomiot. Lisäksi kerrotaan mitä työstä jäi uupumaan sekä pohditaan mahdollisia jatkokehitysideoita.

2 LÄHTÖKOHTA JA TUTKIMUSPROSESSI

Tutkimus lähtee liikkeelle ISO/IEC:n pienille ohjelmistoyksiköille tarkoitetusta standardista, jolla on tutkimuksessa kaksijakoinen rooli. Ensinnäkin on tarkoitus luoda elektroninen prosessiopas, joka perustuisi pienten ohjelmistoyksiköiden prosessistandardiin. Standardia olisi tarkoitus myös noudattaa tutkimusta tehdessä prosessioppaan toteutuksen osalta.

Tässä kohdassa esitellään ISO/IEC:n pienille ohjelmistoyksiköille tarkoitettu prosessistandardi ISO/IEC TR 29110. Tarkoitus on antaa koko prosessistandardista yleiskatsaus, joka perustuu ISO/IEC TR 29110-1:2011(E) -dokumenttiin sekä esitellä hallinnan ja toteutuksen Entry-tason opas ISO/IEC TR 29110-5-1-1:2012(E).

Pienen ohjelmistoyksikön prosessistandardi

Ohjelmistoteollisuus tunnistaa hyvin pienet yksiköt (Very Small Entity, VSE) ja niiden arvon. Hyvin pienissä ohjelmistoyksiköissä (VSE) on ISO/IEC 29110:n mukaan enintään 25 työntekijää. VSE-yksikkönä voivat olla yritys, organisaatio, osasto tai projekti. VSE:den tunnistamista usein vaaditaan, koska ne usein kehittävät ja/tai ylläpitävät suuremmissa järjestelmissä käytettävää ohjelmistoa.

Laajojen kansainvälisten standardien suhteuttaminen hyvin pienten yksiköiden tarpeisiin on yleensä hankalaa ja raskasta. Useimmilla VSE:llä ei ole resursseja perustaa ohjelmiston elinkaariprosesseja, eivätkä koe sitä edes tarpeelliseksi. Näitä ongelmia on pyritty korjaamaan luomalla joukko oppaita, jotka perustuvat muiden standardien osajoukkoihin. Näitä osajoukkoja kutsutaan VSE profiileiksi ja niiden tarkoitus on määritellä VSE asiayhteyteen kansainvälisten standardien sopiva osajoukko. Esimerkiksi ISO/IEC 12207:n prosessit ja työntulokset sekä ISO/IEC 15289:n tuotteet.

ISO/IEC 29110 on kehitetty parantamaan tuotteen ja/tai palvelun laatua ja prosessin suorituskykyä. Taulukossa 2.1 on esitetty ISO/IEC 29110:n osat ja niiden kohdeyleisöt. ISO/IEC 29110:lla ei ole tarkoitus sulkea pois eri elinkaarimallien, kuten esimerkiksi vesiputousmallin, käyttöä.

Taulukko 2.1 - ISO/IEC 29110:n osat ja niiden kohdeyleisöt (ISO/IEC 2011)

ISO/IEC 29110	Title	Target audience
Part 1	Overview	VSEs, customers, assessors, standards producers, tool vendors, and methodology vendors.
Part 2	Framework and taxonomy	Standards producers, tool vendors and methodology vendors. Not intended for VSEs.
Part 3	Assessment guide	Assessors, customers, and VSEs
Part 4	Profile specifications	Standards producers, tool vendors and methodology vendors. Not intended for VSEs.
Part 5	Management and engineering guide	VSEs and customers

Jos tarvitaan uutta VSE profiilia, ISO/IEC 29110-4 ja ISO/IEC TR 29110-5 voidaan kehittää vaikuttamatta olemassa oleviin dokumentteihin. Uudet profiilit ovat muotoa ISO/IEC 29110-4-m tai ISO/IEC 29110-5-m-n.

ISO/IEC TR 29110-5-m-n tarjoaa toteutuksen hallinnan ja suunnittelun opasta ISO/IEC 29110-4-m:ssä kuvattuun VSE-profiilille.

Hallinnan ja suunnittelun Entry-profiilin prosessiopas

ISO/IEC 29110:n osa 5-1-1 tarjoaa toteutuksen hallinnan ja suunnittelun oppaan yleisen profiiliryhmän (Generic Profile Group) Entry-profiilille. Entry-profiilissa kuvataan yksittäisen sovelluksen ohjelmistokehitystä yksittäisellä projektiryhmällä ilman, että on olemassa mitään erityistä riskiä tai tilannekertoimia aloittaville VSE:lle. Aloittavalla VSE:llä tarkoitetaan niitä yksiköitä, jotka ovat aloittaneet toimintansa alle kolme vuotta sitten. Entry-profiilissa voidaan kuvata myös VSE:t, jotka työskentelevät pienten, alle kuusi henkilötyökuukautta, projektien parissa.

Opasta on tarkoitus käyttää minkä tahansa prosessin, tekniikan tai menetelmän kanssa, joka parantaa VSE:n asiakastyytyväisyyttä ja tuottavuutta. Hallinnan ja suunnittelun opasta sovelletaan ohjelmistokehitykselle omistautuneihin yksiköihin. Entry-profiili on yleisen profiiliryhmän ensimmäinen ja kevyin profiili. Opas tarjoaa projektinhallinta ja ohjelmiston toteutus -prosessit (Project Management ja Software Implementation), jotka yhdistävät valittuja paloja ISO/IEC 12207:2008:sta ja ISO/IEC 15289:2011:sta.

VSE:n on tarkoitus käyttää osaa ISO/IEC 29110:sta perustamaan prosesseja kehityksen lähestymistavan tai metodiikan toteutukselle, mukaan lukien esimerkiksi ketterä, evolutionaarinen, inkrementaalinen, testiorientoitunut kehitys jne., jotka pohjautuvat VSE organisaation tai projektin tarpeisiin.

Opasta käyttämällä VSE voi saada seuraavia etuja:

- Asiakkaalle toimitetaan sovittu joukko projektivaatimuksia ja odotettuja tuotteita
- Seurataan systemaattista toteutusprosessia, joka tyydyttää asiakkaan tarpeet ja varmistaa tuotteiden laadun.

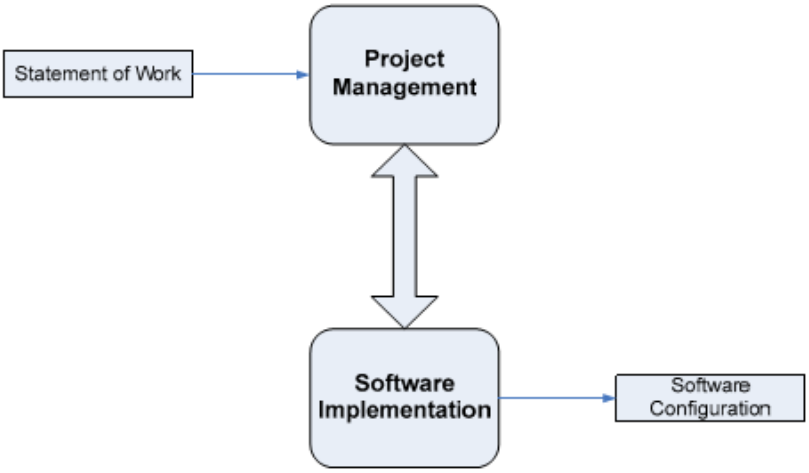
VSE vaatii, että täytetään aloitusehtoja, jotta opasta voidaan käyttää:

- Projektin työnesitys (statement of work) dokumentoidaan
- Osoitetaan pätevä projektiryhmä ja projektipäällikkö
- Projektin aloittamiseen tarvittavat resurssit ja palvelut ovat saatavissa.

Projektinhallinta ja ohjelmiston toteutus

Projektinhallintaprosessin (Project Management, PM) tarkoituksena on perustaa ja toteuttaa ohjelmiston toteutus -prosessin tehtävät järjestelmällisellä tavalla siten, että sallitaan projektin tavoitteiden noudattaminen odotetussa laadussa, ajassa sekä kustannuksessa.

Ohjelmiston toteutus -prosessin (Software Implementation, SI) tarkoitus on analysoinnin, ohjelmistokomponentin tunnistamisen, rakentamisen, integroinnin ja testauksen, sekä tuotteen toimitusaktiviteetin uusien tai muokattujen määriteltyjen vaatimuksien mukaisen ohjelmistotuotteen järjestelmällisen suorituskyvyn saavuttaminen.



Kuva 2.1 - Entry-profiilin prosessit (ISO/IEC TR 29110-5-1-1:2012(E))

Kuvasta 2.1 voidaan huomata, että PM-prosessi käyttää asiakkaan työnesitystä projektisuunnitelman laatimiseen. PM:n projektiarviointi ja -ohjaustehtävät vertaavat projektin kehitystä projektisuunnitelmaa vasten. PM-prosessin sulkemisaktiviteetti toimittaa SI:n tuottaman ohjelmistokokoonpanon ja saa asiakkaan hyväksynnän, jolla virallistetaan projektin päätös.

SI-prosessin suoritusta ohjaa projektisuunnitelma. SI-prosessi alkaa alustusaktiviteetilla, jossa projektisuunnitelma tarkastetaan. Projektisuunnitelma ohjaa ohjelmiston vaatimusanalyysin, ohjelmistokomponentin tunnistuksen, ohjelmiston rakentamisen, ohjelmiston integroinnin ja testauksen, sekä tuotteen toimitusaktiviteettien suoritusta.

Jotta saadaan poistettua tuotteesta vikoja ja puutteita, aktiviteettien työnkulkuun sisältyy todennus-, kelpuutus- ja testitehtäviä. Asiakas toimittaa työesityksen projektinhallintaprosessille ja ohjelmiston toteutus -prosessin tuloksena saadaan ohjelmistokokoonpano (Kuva 2.1).

3 TOTEUTUSTEKNIIKAT

Tässä luvussa esitellään tutkimuksessa käytetyt tekniikat. Kohdassa 3.1 esitellään tiedonhallintajärjestelmä. Kohdassa 3.2 esitellään palvelinpään sovellusalusta. Viimeisessä kohdassa esitellään selainpään työkalupakki.

3.1 Tiedonhallintajärjestelmä

Tässä kohdassa esitellään JSON-tiedonsiirtoformaatti sekä MongoDB-dokumenttitietokanta.

JSON

JSON (JavaScript Object Notation) on kevyt datanvaihtoformaatti. Siitä on tullut ECMA-404 -standardi lokakuussa 2013 ja ennen tätä se on ollut määriteltynä osana JavaScriptia Ecma International ECMA-262 (2011) -standardin 5.1-painosta. Se on täysin ohjelmointikieliriippumaton tekstiformaatti, joka käyttää C-kieliperheestä tuttuja käytäntöjä.

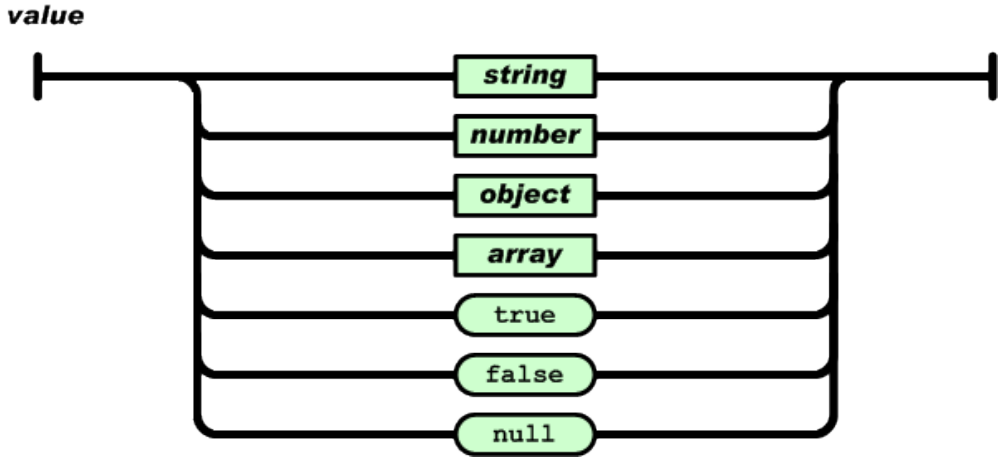
Tässä kohdassa on käytetty lähteenä Ecma International ECMA-404 (2013) -standardin ensimmäistä painosta.

JSON:ssa on kaksi yleisesti käytettyä rakennetta:

- Kokoelma avain/arvo-pareja. Eri kielissä tämä realisoituu objekteina, tietueina, structina, sanakirjoina, hash-tauluina, avainlistana tai assosiaatiotauluna.
- Järjestetty arvolista. Useimmissa kielissä tämä realisoituu taulukkona, vektorina, listana tai sarjana.

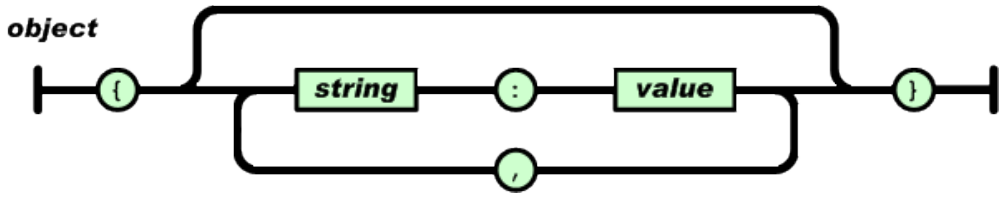
JSON:ssa rakenteet voivat ottaa seuraavat muodot:

Arvo (value, kuva 3.1) voi olla merkkijono lainausmerkeissä, numero, true tai false, null, objekti, tai taulukko. Nämä rakenteet voivat olla sisäkkäisiä.



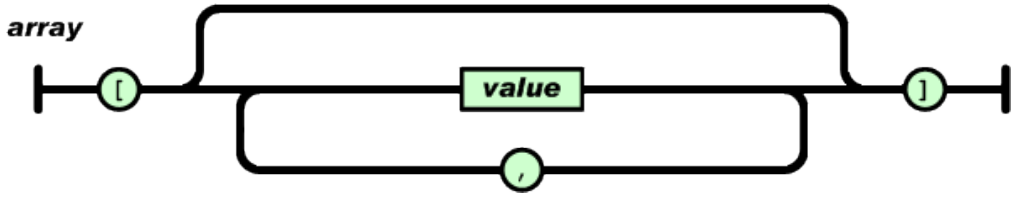
Kuva 3.1 - Arvo (ECMA-404)

Objekti (object, kuva 3.2) on järjestämätön joukko avain/arvo-pareja. Objekti alkaa vasemmalla aaltosulkeella ({) ja päättyy oikeaan aaltosulkeeseen (}). Jokaista avainta seuraa kaksoispiste (:) ja avain/arvo-parit erotetaan pilkulla (,).



Kuva 3.2 - Objekti (ECMA-404)

Taulukko (array, kuva 3.3) on järjestetty kokoelma arvoja. Taulukko alkaa vasemmalla hakasulkeella ([) ja päättyy oikeaan hakasulkeeseen (]). Arvot erotetaan pilkulla (,).



Kuva 3.3 - Taulukko (ECMA-404)

Merkkijono (string) on n-kappaleen sarja Unicode-merkkejä, jossa on mahdollista käyttää kenoviivasuojausta (backslash escape). Merkki (char) esitetään yksittäisenä merkkijonona. Merkkijono vastaa hyvin pitkälle C tai Java -merkkijonoa.

Luku (number) on hyvin samantyyppinen kuin C:n tai Javan luku sillä poikkeuksella, että oktaaliluku- ja heksadesimaalilukuformaatteja ei käytetä.

Esimerkiksi JavaScriptissä JSON:t esitetään objekteina ja ne voivat koostua useista eri tietotyypeistä:

```
{"greeting" : "Hello, world!"}
```

Tämä JSON-notaation mukainen dokumentti sisältää yhden avaimen "`greeting`" ja sitä vastaavan arvon "`Hello world!`". Dokumentit voivat olla myös monimutkaisempia ja sisältää useita avain/arvo-pareja:

```
{"greeting" : "Hello, world!", "foo" : 3}
```

Dokumentit voivat koostua useista eri tietotyypeistä. Ylläolevasta esimerkissä `greeting`-avaimen arvo on merkkijono ja `foo`-avaimen arvo in kokonaisluku.

MongoDB

MongoDB on yleiskäyttöinen tietokanta, joka on samalla tehokas, joustava ja skaalautuva. Sen ominaisuuksiin lukeutuvat muiden muassa kyky skaalautua horisontaalisesti (datan hajauttaminen usealle laitteelle), toissijaiset indeksit, rajatut haut, järjestämiset, koostamiset ja paikkatietoindeksit (geospatial).

Tässä kohdassa on käytetty lähteenä Chodorowin (2013) MongoDB opasta. Ja tarkoitus on esitellä MongoDB:n ominaisuuksia, joiden tuntemisella pääsee alkuun tekniikan käytössä.

Dokumentti vs. relaatio

MongoDB on dokumenttitietokanta ja ero relaatiomalliin on muiden muassa, se että relaatiomallin rivit on korvattu joustavammalla dokumenttimallilla. Kun käytetään sisällytettyjä dokumentteja taulukoissa, dokumenteilla voidaan esittää monimutkaisia hierarkisia rakenteita yhdellä tietueella.

Dokumenttimallissa ei myöskään ole käytössä ennaltamäärättyjä skeemoja, mikä tarkoittaa, ettei dokumentin avain/arvo-pareilla ole mitään ennaltamäärättyjä tyyppejä tai kokoja. Kun skeemaa ei ole kiinnitetty, kenttien lisäämisestä ja poistamisesta tulee yksinkertaisempaa. Tämä aiheuttaa sen, että iteroinnista tulee nopeampaa ja datamalleja voidaan kokeilla ripeämmällä tahdilla.

Skaalautuvuus

Sovelluksien tiedostokoot kasvavat kokoajan. Nopean laajakaistan ja muistien halpeneminen ovat luoneet ympäristön, missä pienienkin sovellusten täytyy tallettaa enemmän dataa kuin monet tietokannat on tarkoitettu käsittelemään. Teratavullinen dataa on nykyisin jo arkipäivää.

Kun tallennettavan datan määrä kasvaa, voidaan valita vertikaalinen (scale up) tai horisontaalinen (scale out) skaalaaminen. Vertikaalinen skaalaus tapahtuu hankkimalla isompi ja tehokkaampi laite nykyisen tilalle. Horisontaalisessa skaalaamisessa hankitaan lisää laitteita ja hajautetaan data usealle laitteelle. Yleensä isompien laitteiden hankinta on helpompaa, mutta ne ovat usein kalliita ja jossakin vaiheessa saavutetaan fyysiset rajoitukset sille kuinka iso ja tehokas laite voi olla. Vaihtoehtona on hankkia toinen palvelin ja liittää se klusteriin. Näin saadaan yleensä edullisemmin lisää tallennustilaa tai suorituskykyä. Ongelmana on puolestaan usean koneen ylläpito.

Koska MongoDB suunniteltiin horisontaaliseen skaalautuvuuteen, sen dokumenttipohjainen datamalli tekee Chodorowin (2013) mukaan datan hajauttamisesta useammalle laitteelle yksinkertaista. MongoDB huolehtii automaattisesti datan ja kuorman tasapainottamisesta palvelinklusterissa, ohjaamalla käyttäjäpyynnöt oikeille laitteille, sekä hajauttamalla dokumentteja. Kun tarvitaan lisää kapasiteettia, voidaan klusteriin lisätä uusia laitteita ja antaa MongoDB:n huolehtia siitä miten data hajautetaan niille.

Ominaisuudet

MongoDB on tarkoitettu olemaan yleiskäyttöinen tietokanta, ja datan luonnin, lukemisen, päivittämisen sekä poistamisen lisäksi, se tarjoaa muiden muassa seuraavia ominaisuuksia:

- **Indeksointi**. MongoDB tukee geneerisiä toissijaisia indeksejä, jotka mahdollistavat erilaiset nopeat kyselyt ja tarjoaa myös uniikki-, yhdistelmä-, geospatiaali- ja teksti-indeksointi mahdollisuudet.

- **Kooste**. MongoDB tukee niin sanottua koosteputkea (aggregation pipeline), joka mahdollistaa monimutkaisten koosteiden rakentamisen palasista.

- **Erikoiskokoelmat**. MongoDB tukee elinaikakokoelmia sellaiselle datalle, jonka pitäisi erääntyä tietyn ajan päästä, kuten sessiot. Se tukee myös määrämittaisia kokoelmia tuoreelle datalle, kuten rekistereille.

- **Tiedostomuisti**. MongoDB tukee isojen tiedostojen ja tiedostojen metadatan tallettamista.

MongoDB:ssa ei ole joitain relaatiotietokannoille tyypillisiä ominaisuuksia, kuten liitoksia ja monen rivin transaktioita. Nämä ominaisuudet on jätetty tietoisesti pois, koska kummankin toteutus hajautetussa ympäristössä on hankalaa.

Nopeus

MongoDB:n merkittävänä tavoitteena on korkea suorituskyky ja tämä on vaikuttanut sen suunnitteluun. Johdonmukaista suorituskykyä pidetään yllä lisäämällä dynaamista täytettä dokumentteihin ja esivaraamalla tiedostoja. RAM:a pyritään varaamaan niin paljon kuin mahdollista ja samalla pyritään automaattisesti valitsemaan kyselyille oikeat indeksit.

Vaikka MongoDB pyrkii tukemaan useita relaatiojärjestelmien ominaisuuksia, sitä ei ole tarkoitettu toimimaan relaatiotietokannan tavoin. Tietokantapalvelin pyrkii siirtämään prosessoinnin ja logiikan asiakassovellukseen ajurien tai käyttäjän sovelluskoodin avulla aina kun on mahdollista.

Peruskäsitteet:

MongoDB:n peruskäsitteitä ovat:

- Dokumentti on datan perusyksikkö MongoDB:ssa ja tarkoittaa karkeasti samaa kuin rivi relaatiotietokantajärjestelmässä, mutta se on paljon ilmaisuvoimaisempi.

- Kokoelma on joukko dokumentteja. Jos dokumentti MongoDB:ssa on vastine riville relaatiotietokannassa, tällöin kokoelmaa voidaan ajatella taulukon vastineena.

- MongoDB:n yksi instanssi voi isännöidä useita itsenäisiä tietokantoja, joista jokaisella voi olla omat kokoelmansa.

- Jokaisella dokumentilla kokoelman sisällä on uniikki `_id` -avain.

- MongoDB:n mukana tulee JavaScript -shell, jolla voidaan hallita MongoDB:n instansseja ja manipuloida dataa.

Dokumentit ja kokoelmat

Dokumentti on järjestetty avaimien ja siihen liittyvien arvojen sarja. Dokumentin esitystapa vaihtelee ohjelmointikielen mukaan, mutta useimmissa kielissä data sopii rakenteeseen luonnollisesti, kuten map, hash tai sanakirja. Esimerkiksi JavaScriptissä dokumentit esitetään objekteina ja ne voivat koostua useista eri tietotyypeistä.

Dokumentin avaimet ovat merkkijonotyyppisiä. Mikä tahansa UTF-8 merkki on sallittu avaimessa, ainoat poikkeukset ovat null-merkki (\0), sekä piste ja dollari -merkit.

Null:lla merkitään avaimen loppu, ja piste ja dollari -merkit on tarkoitettu edistyneisiin toimintoihin.

MongoDB:ssa isot ja pienet kirjaimet sekä tyypit on eroteltu. Esimerkiksi nämä dokumentit eivät ole samoja:

```
{"foo" : 3}
{"foo" : "3"}
```

eivätkä nämä:

```
{"foo" : 3}
{"Foo" : 3}
```

MongoDB:n dokumenteissa ei voi olla samoja avaimia, vaikka tämä on JSON-notaatiossa sallittu. Esimerkiksi seuraava dokumentti ei ole sallittu:

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

Dokumenttien avain/arvo-parit ovat järjestettyjä: {“x” : 1, ”y” : 2} ei ole sama asia kuin {”y” : 2, ”x” : 1}. MongoDB voi järjestää parit uudelleen ja tästä syystä skeemoja ei pitäisi suunnitella olemaan riippuvaisia parien järjestyksestä.

Dynaamiset skeemat

Kokoelmilla on dynaamiset skeemat. Tämä tarkoittaa sitä, että kokoelman sisällä olevien dokumenttien “muodot” voivat olla minkälaisia tahansa. Esimerkiksi molemmat näistä dokumenteista voitaisiin tallettaa samaan kokoelmaan:

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

Mikä tahansa dokumentti voidaan asettaa mihinkä kokoelmaan tahansa:

- Erilaisten dokumenttien pitäminen samassa kokoelmassa voi aiheuttaa lisätyötä kehittäjille ja ylläpitäjille. Kehittäjien täytyy varmistaa, että jokainen kysely palauttaa vain tietyn tyyppisiä dokumentteja tai kyselyä suorittava sovelluskoodi selviytyy erimuotoisista dokumenteista.

- On paljon nopeampaa noutaa kokoelmien lista, kuin kerätä lista kokoelmassa olevista tyypeistä. Esimerkiksi, olisi paljon hitaampaa etsiä kolme arvoa yhdestä kokoelmasta kuin, että olisi kolme kokoelmaa, jotka haettaisiin kyselyllä.

- Samankaltaisten dokumenttien niputtaminen samaan kokoelmaan mahdollistaa datan paikkakohtaisuuden, jolloin levyhakuja tarvitaan todennäköisesti vähemmän.

- Kun dokumenteille luodaan indeksejä, niille alkaa määräytyä rakenne. Nämä indeksit määräytyvät kokoelman mukaan. Kokoelmat voidaan indeksoida tehokkaammin, jos laitetaan vain tietyn tyyppisiä dokumentteja samaan kokoelmaan.

MongoDB ei pakota yhteenkuuluvien dokumenttien niputtamista samaan kokoelmaan. Kokoelma tunnistetaan nimensä mukaan. Kokoelman nimi voi olla, samoilla poikkeuksilla kuin dokumentissa, mikä tahansa UTF-8 merkkijono.

Tietokannat

MongoDB:ssa ryhmitellään kokoelmat tietokannoiksi. MongoDB:n instanssi voi isännöidä useaa tietokantaa, joista jokaisella on omat käyttöoikeutensa. Tietokannat on talletettu levyllä erillisiin tiedostoihin ja muun muassa tästä syystä saman sovelluksen kaikki tieto olisi hyvä tallettaa samaan tietokantaan.

Tietokannat tunnistetaan nimen perusteella. Nimi voi olla UTF-8 -merkkijono samoilla rajoituksilla kuin on dokumentilla ja kokoelmalla. Koska tietokannat päätyvät tiedostoiksi tiedostojärjestelmään, aiheutuu tästä lisää rajoituksia nimeämiselle.

MongoDB:ssa on useita varattuja tietokantanimiä:

- Admin (`admin`) on tunnistautumisen juuri-tietokanta. Admin-tietokantaan lisätty käyttäjä perii automaattisesti oikeudet kaikkiin tietokantoihin. Admin-tietokannasta voidaan myös ajaa koko palvelimen laajuisia komentoja, kuten kaikkien tietokantojen listaus tai palvelimen sammuttaminen.

- Local-tietokantaa (`local`) ei koskaan kahdenneta ja mikä tahansa kokoelma voidaan tallettaa tänne, mutta se pysyy paikallisena yhdellä koneella.

- Mongo käyttää `config`-tietokantaa tallettaakseen tietoa järjestelmän eri sirpaleista (shard), kun sitä käytetään hajautetussa järjestelmässä.

Perustietotyypit

MongoDB:n dokumentteja voidaan ajatella JSON-tyyppisinä, sillä ne ovat käsitteellisesti samanlaisia JavaScript objektien kanssa. JSONin ilmaisuvoima on kuitenkin rajoittunut, koska ainoat käytössä olevat tyypit ovat null, boolean, numeric, string, array ja object.

MongoDB lisää tukea usealle tietotyypille, jotka puuttuvat JSON-formaatista:

- **Null**. Nullia voidaan käyttää null-arvon sekä olemattoman kentän esittämiseen:
```
{"x" : null}
```

- **Boolean**. On olemassa looginen tyyppi, jota voidaan käyttää true ja false -arvoihin:
```
{"x" : true}
```

- **Numero**. Shell käyttää oletusarvoisesti 64-bittisiä liukulukuja. Tästä syystä nämä numerot näyttävät ns. normaaleilta shellissä:
```
{"x" : 3.14}
```
tai:
```
{"x" : 3}
```

  Kokonaislukujen osalta tulisi käyttää NumberInt tai NumberLong -luokkia, jotka edustavat 4- tai 8-bittisiä etumerkillisiä kokonaislukuja.
```
{"x" : NumberInt("3")}
{"x" : NumberLong("3")}
```

- **Merkkijono**. Mikä tahansa UTF-8 -merkkijono voidaan esittää käyttämällä merkkijonotyyppiä string:
```
{"x" : "foobar"}
```

- **Päivämäärä**. Päivämäärät talletetaan epookki-millisekunteina. Aikavyöhykettä ei talleteta:
```
{"x" : new Date()}
```

- **Säännöllinen lauseke**. Säännöllisiä lausekkeita voidaan käyttää kyselyissä käyttämällä JavaScriptin säännöllisien lausekkeiden syntaksia:
```
{"x" : /foobar/i}
```

- **Taulukko**. Joukkoja tai listoja voidaan esittää taulukkoina:
```
{"x" : ["a", "b", "c"]}
```

- **Sisällytetty dokumentti**. Dokumentit voivat sisältää toisia dokumentteja arvoina:
```
{"x" : {"foo" : "bar"}}
```

- **Object id**. Objektitunniste on 12-tavuinen tunniste dokumenteille.
```
{"x" : ObjectId()}
```

Harvinaisempia tyyppejä, joita voidaan tarvita:

- **Binäärinen data**. Binäärinen data on mielivaltaisten tavujen merkkijono. Sitä ei voida manipuloida shellistä käsin. Ainoa tapa tallettaa ei-UTF-8 merkkijonoja tietokantaan on binäärinen data.

- **Koodi**. Kyselyt ja dokumentit voivat myös sisältää JavaScript koodia:

```
{"x" : function() { /* ... */ }}
```

Taulukot

Taulukot ovat arvoja, joita voidaan käyttää sekä järjestetyissä (listat, pinot ja kyselyt) että järjestämättömissä (joukot) operaatioissa.

Seuraavassa dokumentissa avaimella `things` on arvona taulukko:

```
{"things" : ["pie", 3.14]}
```

Yllä olevasta esimerkistä nähdään, että taulut voivat sisältää erilaisia datatyyppejä arvoinaan (tässä merkkijono ja liukuluku). Taulukkojen arvot voivat olla myös sisäkkäisiä tauluja.

MongoDB ymmärtää dokumenteissa olevien taulujen rakenteen ja pystyy suorittamaan operaatioita taulujen sisällöllä. Tämä mahdollistaa kyselyjen tekemisen tauluihin ja indeksien rakentamisen sisällöstä. Esimerkiksi edellisessä esimerkissä MongoDB pystyy tekemään kyselyn kaikkiin dokumentteihin, missä `3.14` on `things`-taulukon elementti. Voidaan luoda indeksi `things`-avaimelle nopeuttamaan kyselyä, jos tietoa haetaan usein.

MongoDB sallii taulukkojen arvoja muuttavia atomisia päivityksiä, kuten esimerkiksi muuttamalla taulun arvo `pie` muotoon `pi`.

Sisäkkäiset dokumentit

Sisäänrakennettujen dokumenttien avulla voidaan järjestää dataa luonnollisemmin kuin litteän rakenteen avain/arvo-pareilla. Esimerkiksi, jos on henkilöä esittävä dokumentti ja halutaan tallettaa henkilön osoite, voidaan tieto sisällyttää sisäänrakennettuun `address`-dokumenttiin:

```
{
"name" : "John Doe",
"address" : {
             "street" : "123 Park Street",
             "city" : "Anytown",
             "state" : "NY"
             }
```

```
}
```

Edellisen esimerkin `address`-avaimen arvo on sisäänrakennettu dokumentti omilla avain/arvo-pareillaan.

Esimerkin dokumentti mallinnettaisiin relaatiotietokannassa kahdeksi eri riviksi kahdessa eri taulussa (yksi taulu henkilöille ja yksi osoitteille). MongoDB:lla osoitedokumentti voidaan sisällyttää suoraan henkilö-dokumentin sisälle. Sisällytetyt dokumentit voivat tarjota luonnollisemman esitystavan tiedoille.

Kääntöpuolena MongoDB:ssa on datan toistaminen. Oletetaan, että osoitteet olisivat erillinen taulu relaatiotietokannassa ja eräästä osoitteesta pitäisi korjata kirjoitusvirhe. Kun tehdään henkilö- ja osoite -taulujen liitos, saadaan päivitetty osoite kaikille, jotka sitä käyttävät. MongoDB:ssa kirjoitusvirhe täytyisi korjata jokaisen henkilön dokumentista.

_id ja ObjectID

Oletusarvoisesti _id-avain on ObjectId-tyyppiä, mutta se voidaan asettaa miksi tahansa tyypiksi. Jokaisella dokumentilla täytyy olla uniikki arvo _id-avaimelle saman kokoelman sisällä, joka varmistaa että jokainen dokumentti on yksilöllisesti tunnistettavissa.

ObjectId on generoitavissa globaalisti uniikilla tavalla eri laitteilla. MongoDB:n jaettu luonne on pääsyy sille, miksi käytetään ObjectId:tä, eikä esimerkiksi automaattisesti kasvavaa perusavainta. Automaattisesti kasvavan perusavaimen synkronointi usealla palvelimella on aikaa vievää ja hankalaa. Lisää tietoa ObjectId-avaimen luomisesta löytyy liitteestä 2.

Jos _id-avainta ei ole, kun dokumenttia lisätään, se lisätään automaattisesti dokumenttiin. MongoDB-palvelin voi hoitaa tämän, mutta yleensä tämä annetaan ajurin hoidettavaksi asiakasohjelman puolella. Päätös luoda ne asiakasohjelman puolella heijastaa MongoDB:n yleisfilosofiaa: työ pitäisi siirtää pois palvelimelta ajureille aina kuin mahdollista. Tämä filosofia kielii siitä, että jopa skaalautuvien tietokantojen kanssa on helpompaa skaalata horisontaalisesti sovellustasolla kuin tietokantatasolla. Kuormaa saadaan vähennettyä, kun työ siirretään asiakasohjelmalle.

3.2 Palvelinpään sovellusalusta

Tässä kohdassa esitellään Node.js ja siihen ladattavia moduuleja, sekä REST-arkkitehtuurityyli.

Node.js

Tässä kohdassa on käytetty lähteenä Teixeiran (2013) laatimaa opasta ohjelmistokehitykselle Node.js-alustalla.

Node.js (jäljempänä Node) on Googlen V8 JavaScript-moottorin, tapahtumasilmukan ja matalan tason I/O:n alusta. Node tarjoaa tapahtumapohjaisen ja rinnakkaisesti toimivan alustan verkko-ohjelmiston rakentamiselle JavaScript-kielellä.

Node on ollut alan suurimpien toimijoiden huomion keskipisteenä ja muiden muassa Microsoftin Azure-pilvipalvelua voidaan käyttää yhdessä Noden kanssa. Noden etuja on JavaScriptin käyttö, koska se on yksi laajimmin käytetty ohjelmointikieli ja web-kehittäjät ovat tottuneet kirjoittamaan JavaScriptia selaimelle, tuntuu luonnolliselta, että myös palvelimelle voisi ohjelmoida samalla kielellä. Node siis mahdollistaa ohjelmoinnin sekä selain- että palvelinpäähän samalla kielellä.

Node on pelkkä alusta, johon kaikki pitää itse rakentaa. Tämä tarkoittaa, että Noden ydintoiminnallisuudet on pidetty minimissä ja monimutkaisuutta on pyritty välttämään. Nodessa on jo itsessään valmiita komponentteja yksinkertaisten sovellusten rakentamiseen, mutta kun otetaan käyttöön kolmannen osapuolen moduuleja, voidaan toteuttaa monimutkaisiakin sovelluksia.

Node Package Manager

Noden asennuksen mukana tulee Noden pakettienhallintatyökalu (NPM), joka on kolmannen osapuolen pakettiarkisto. Se on myös tapa hallita tietokoneelle asennettuja paketteja, se on myös tapa määritellä riippuvuudet muihin paketteihin. NPM tarjoaa julkisen rekisteripalvelun, joka sisältää kaikki ohjelmoijien NPM:ssä julkaisemat paketit. NPM tarjoaa myös komentorivityökalun näiden pakettien lataamiseen, asentamiseen ja hallintaan.

NPM ylläpitää julkisten moduulien keskitettyä kirjastoa ja sitä voidaan selata osoitteessa https://www.npmjs.org/. Noden avoimen lähdekoodin moduulin kirjoittaja voi halutessaan julkaista moduulin NPM:iin, ja asennusohjeissa tulisi olla NPM moduulin nimi, jonka avulla se voidaan ladata ja asentaa.

NPM peruskäyttötarkoitukset ovat pakettien asentaminen ja poistaminen. Tähän liittyvät NPM:n kaksi päätoimintatilaa: globaali ja paikallinen. Nämä kaksi tilaa vaihtavat pakettien tallennushakemistojen kohdetta ja vaikuttavat siihen kuinka Node lataa moduuleja.

NPM:ssa paikallinen tila on oletuksena päällä. Tässä tilassa NPM toimii paikallisella hakemistotasolla, eikä tee järjestelmän laajuisia muutoksia. Tässä tilassa asennetut mo-

duulit eivät vaikuta muihin paikallisesti asennettuihin sovelluksiin. Globaali tila sopii paremmin moduulien asentamiselle, jotka tulisi aina olla saatavilla kaikille Node-sovelluksille.

Paikallista tilaa tulisi käyttää oletuksena ja jos moduulin kirjoittajat haluavat moduulin globaalia asennusta, siitä yleensä kerrotaan asennusohjeissa. Paikallisessa tilassa voidaan valita sovelluskohtaisesti, mitä moduuleja ja versioita otetaan käyttöön, ilman että häiritään globaalia moduuliavaruutta. Tämä tarkoittaa sitä, että kaksi sovellusta voi olla riippuvainen saman moduulin eri versioista, eivätkä ne aiheuta ristiriitaa keskenään. Paikallisessa tilassa NPM toimii `node_modules` -hakemistossa nykyisen työhakemiston alla.

Express

Express.js on kuvattu minimaaliseksi ja joustavaksi Node -web-sovelluskehykseksi, joka tarjoaa joukon ominaisuuksia yhden ja useamman sivun tai hybridi -web-sovelluksia. Se on ohjelmistokehys Node.js -web-sovellusten rakentamiselle. Express tarjoaa työkalu HTTP-palvelimille, joka tekee siitä hyvän ratkaisun yhden sivun sovelluksille, web-sivuille, hybrideille tai julkisille HTTP API:lle. (NPM 2014a, Express 2014)

MongoDB Driver

MongoDB Node.js Native Driver on MongoDB:n virallinen Node-ajuri, joka mahdollistaa MongoDB:n käyttämisen Node-sovelluksien kanssa. (NPM 2014b)

REST-arkkitehtuurityyli

REST määrittelee joukon arkkitehtuurillisia periaatteita, joilla voidaan suunnitella web-palveluita. Ne keskittyvät järjestelmän resursseihin, mukaan lukien siihen, miten resurssitilat osoitetaan ja siirretään HTTP-protokollaa käyttämällä laajalla valikoimalla eri kielillä kirjoitettuja asiakasohjelmia. REST on noussut muutamassa vuodessa noussut vallitsevaksi web-palvelujen suunnittelumalliksi, jos lasketaan sitä käyttävien web-palvelujen määrä. Itse asiassa RESTillä on ollut niin suuri vaikutus webiin, että se on suurimmaksi osaksi korvannut SOAP- ja WSDL-pohjaiset rajapintojen suunnittelumallit, koska sen käyttäminen on yksinkertaisempaa.

Tämä kohta perustuu kokonaisuudessaan Rodriguezin (2008) kirjoittamaan oppaaseen Representational State Transfer -arkkitehtuurityylin (REST) mukaisen web-palvelun perusteista.

REST:n web-palvelun toteutus seuraa neljää perussuunnitteluperiaatetta:

- HTTP-metodeja tulisi käyttää eksplisiittisesti.
- Sen tulisi olla tilaton (stateless).
- URIt tulisi näyttää hakemistokaltaisina.
- Tiedonsiirtoformaatti tulisi olla XML, JSON tai molemmat.

Rodriguezin (2008) mukaan REST ei saanut näin paljon huomiota 2000, kun Roy Fielding (2000) esitteli sen väitöskirjassaan ”Architectural Styles and the Design of Network-based Software Architectures”, joka analysoi joukkoa ohjelmistoarkkitehtuuriperiaatteita, jotka käyttävät webiä hajautetun tietojenkäsittelyn alustana. Vuosia myöhemmin merkittäviä ohjelmistokehyksiä RESTille on ilmestynyt ja niitä kehitetään yhä. Viimeisimpinä muun muassa Node.js-alustalle saatava Express, jolla voidaan toteuttaa RESTin toiminnallisuutta muutamalla rivillä JavaScript-koodia.

HTTP-metodit

Yksi RESTful web-palvelun keskeisistä tuntomerkeistä on HTTP-metodien käyttö HTTP-protokollan mukaisesti, joka on määritelty IETF RFC 2616 (1999):ssa, jonka yhtenä kirjoittajana Fielding on myös toiminut. RFC 2616 on kuitenkin merkitty vanhentuneeksi kesäkuussa 2014 ja Fielding on myös ollut kirjoittamassa sen korvaavia IETF-standardeja: RFC7230, RFC7231, RFC7232, RFC7233, RFC7234, RFC7235.

Esimerkiksi HTTP GET määritellään dataa tuottavaksi metodiksi, jota asiakasohjelmiston on tarkoitus käyttää resurssin tai datan noutamiseen web-palvelimelta, tai kyselyn ajamiseen sillä odotusarvolla, että web-palvelin hakee ja palauttaa joukon kyselyä vastaavia resursseja.

REST vaatii kehittäjiä käyttämään HTTP-metodeja yhdenmukaisesti HTTP-protokollan kanssa. Tämä RESTin perussuunnitteluperiaate laatii yksi yhteen vastaavuuden luonti-, luku-, päivitys- ja poisto-operaatioiden (create, read, update, delete - CRUD) sekä HTTP-metodien välille. Seuraavalla tavalla:

- Resurssin luomiseksi palvelimella käytetään POST-metodia.
- Resurssin noutamiseen käytetään GET-metodia.
- Resurssin tilan muuttamiseen tai sen päivittämiseen käytetään PUT-metodia.
- Resurssin poistamiseen käytetään DELETE-metodia.

HTTP-metodien käyttäminen siellä, missä niitä ei ole tarkoitettu, on suunnitteluvirhe, joka on periytynyt moneen web-ohjelmointirajapintaan. Pyydettävän resurssin tunniste (the request URI) HTTP GET -pyynnössä määrittelee yleensä yhden spesifisen resurssin. Kyselymerkkijono pyydettävän resurssin tunnisteessa sisältää joukon parametreja, jotka määrittelevät palvelimen käyttämät etsintäkriteerit osuvien resurssien löytämiselle.

Ainakin tällä tavoin HTTP/1.1 RFC kuvailee GET-metodin. Mutta on useita web-ohjelmointirajapintatapauksia, joissa käytetään HTTP GET-metodia laukaisemaan jotakin transaktionaalista palvelimella, esimerkiksi tietueiden lisääminen tietokantaan. Näissä tapauksissa GET-pyynnön resurssitunnistetta ei käytetä oikein tai ei ainakaan täyden REST:n (RESTfully) kaltaisesti. Jos web-ohjelmointirajapinta käyttää GET-metodia herättämään etäproseduureja, näyttää se tältä:

```
GET /adduser?name=Robert HTTP/1.1
```

Rakenne ei ole kovin hyvä, koska yllä oleva web-metodi tukee tilan vaihtavia operaatioita HTTP GET -metodin yli. Toisin sanoen, yllä olevalla HTTP GET -metodilla on sivuvaikutuksia. Pyynnön tuloksen on määrä lisätä uusi käyttäjä, tässä esimerkissä Robert, taustalla olevaan tietovarastoon, jos pyynnön käsittely suoriutuu onnistuneesti. Ongelma tässä on pääasiassa semanttinen. Web-palvelimet on suunniteltu vastaamaan HTTP GET -pyyntöihin noutamalla resurssit, jotka vastaavat pyydettävän resurssin tunnisteen polkua (tai kyselyn kriteerejä) ja palauttavat nämä tai esityksen vastauksesta, eikä lisäämään tietuetta tietokantaan. Sekä protokolla-metodin aiotun käytön näkökulmasta että HTTP/1.1-yhteensopivien web-palvelimien näkökulmasta, GET-metodin tämän tyyppinen käyttö on epäjohdonmukaista.

Semantiikan taustalla toinen ongelma GET-metodin kanssa on tietueen poistamisen, muuttamisen tai lisäämisen liipaisu tietokannassa, tai palvelinpuolen tilan muuttaminen jollakin tapaa. Tämä kutsuu web-välimuistityökaluja (hakurobotteja - crawlers) ja hakumoottoreita tekemään palvelinpuolen muutoksia vahingossa hakurobotin käsitellessä linkkiä. Tästä ongelmasta voi suoriutua yksinkertaisesti siirtämällä pyydettävän resurssin tunnisteessa olevat parametrien nimet ja arvot XML-tunnisteiden sisään. Syntyvät tunnisteet (XML-esitys luotavasta oliosta) voidaan lähettää HTTP POST -metodin body-osassa, jonka pyydettävä resurssin tunniste (request URI) on olion tarkoitettu isäntäelementti (parent) (katso listaukset 3.1 ja 3.2).

Listaus 3.1 - Ennen

```
GET /adduser?name=Robert HTTP/1.1
```

Listaus 3.2 - Jälkeen

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
    <name>Robert</name>
</user>
```

Yllä oleva menetelmä on esimerkillinen tapaus RESTful-pyynnöstä: HTTP POST -metodin oikeaoppinen käyttö ja tietosisältö on sisällytetty pyynnön runkoon. Vastaanot-

topäässä pyyntöä voidaan käsitellä lisäämällä rungossa oleva resurssi resurssitunnisteessa ilmoitetun resurssin alaiseksi. Tässä tapauksessa uusi resurssi tulisi lisätä /users lapsielementiksi. Tämäntyyppinen sisältyvyyssuhde uuden olion ja sen isäntäolion (parent) välillä, (kuten on määritelty POST-metodissa) vastaa sitä, miten tiedosto on isäntähakemiston alainen. Asiakaspää (client) asettaa olion ja sen isännän välisen suhteen, sekä määrittelee uuden olion URI:n POST-pyynnössä.

Asiakassovellus voi seuraavaksi saada resurssin esitysmuodon käyttämällä uutta URI:a, huomaten, että resurssi sijaitsee ainakin loogisesti `/users` alla, kuten listauksessa 3.3.

Listaus 3.3 - HTTP GET -pyyntö

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

GETin käyttäminen tällä tavoin on yksiselitteistä, koska GET on tarkoitettu ainoastaan datan noutamista varten. GET on operaatio, jolla ei pitäisi olla sivuvaikutuksia, tämä ominaisuus tunnetaan myös nimellä idempotenssi.

GET-metodin samanlaista uudelleen järjestelyä tarvitaan myös tapauksissa, joissa päivitysoperaatiota tuetaan HTTP GET -metodia käyttämällä, kuten listauksessa 3.4.

Listaus 3.4 - päivitys HTTP GETiä käyttämällä

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

Tämä muuttaa resurssin nimiattribuuttia (tai ominaisuutta). Vaikka kyselymerkkijonoa voidaan käyttää sellaiseen operaatioon, tämänlaisella merkkijono-metodin-signatuurina -mallilla (query-string-as-method-signature pattern) on tapana hajota, kun sitä käytetään monimutkaisemmissa operaatioissa. Koska tavoitteena on käyttää HTTP-metodeja yksiselitteisesti, RESTful-tyyppisempi lähestymistapa on lähettää HTTP PUT -pyyntö päivittämään resurssia HTTP GET -metodin sijaan (listaus 3.5).

Listaus 3.5 - HTTP PUT -pyyntö

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
    <name>Bob</name>
</user>
```

Alkuperäisen resurssin korvaaminen PUT-metodilla tarjoaa puhtaamman rajapinnan, joka on yhdenmukainen REST:n periaatteiden ja HTTP-metodien määritelmien kanssa. PUT-pyyntö listauksessa 3.5 on yksiselitteinen siinä mielessä, että se osoittaa päivitet-

tävän resurssin tunnistamalla sen pyydettävän resurssin tunnisteessa, ja siirtää resurssin uuden esityksen asiakasohjelmistolta palvelimelle PUT-pyynnön rungossa, sen sijaan, että resurssiattribuutit siirrettäisiin löyhänä joukkona parametrien nimiä ja arvoja pyydettävän resurssin tunnisteessa. Listauksessa 3.5 nimetään resurssi uudelleen `Robertista Bob`iksi ja tämä muuttaa URI:n muotoon `/users/Bob`. REST web-palvelussa vanhaan URI:n tehdyt pyynnöt aiheuttavat standardin mukaisen 404 Not Found -virheen.

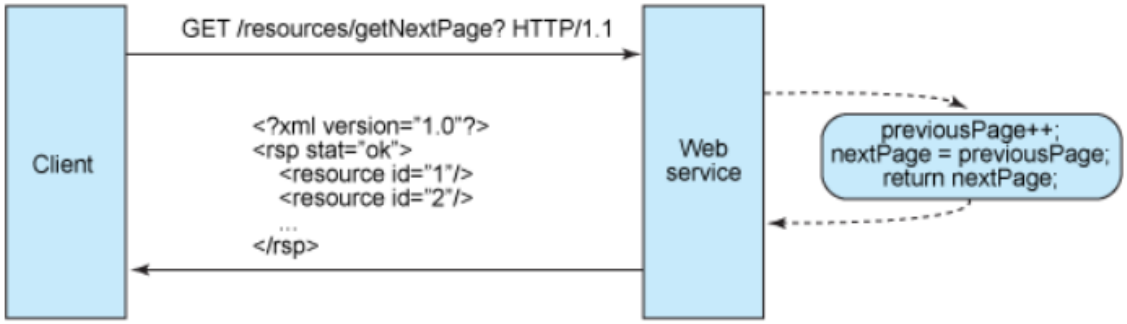
Yleisenä suunnitteluperiaatteena auttaa, kun seuraa REST:n suuntaviivoja HTTP-metodien käytöstä yksiselitteisesti käyttämällä URIssa substantiiveja verbien sijasta. RESTful web-palvelussa verbit POST, GET, PUT ja DELETE on jo määritelty protokollan toimesta. Ihannetapauksessa, jotta voidaan pitää rajapinta yleisluontoisena ja asiakaspuoli voi kutsua yksiselitteisesti operaatioita, web-palveluun ei pitäisi määritellä enemmän verbejä tai etäproseduureja, kuten `/adduser` tai `/updateuser`. Tämä yleinen rakenneperiaate pätee myös HTTP-pyynnön runkoon, joka tarkoitettu siirtämään resurssin tila, eikä kuljettamaan etämetodin nimeä tai etäproseduuria kutsuttavaksi.

Tilattomuus

REST:n web-palveluiden tarvitsee olla skaalautuvia vastatakseen yhä kasvavaan korkean suorituskyvyn kysyntään. Palvelinklusterien kuormantasaus ja failover-proseduuri kyvyillä, välityspalvelimet, sekä yhdyskäytävät ovat tyypillisesti järjestetty siten, että ne muodostavat palvelutopologian, joka sallii tarvittaessa pyyntöjen välittämisen palvelimelta toiselle pienentämään web-palvelukutsun vasteaikaa. Välittäjäpalvelimien käyttäminen skaalautuvuuden parantamiseen vaatii, että REST:n web-palveluiden asiakasohjelma lähettää kokonaisia (complete) itsenäisiä pyyntöjä. Eli lähettää pyyntöjä, jotka sisältävät kaiken tarvittavan datan ilman, että sitä pidettäisiin paikallisesti.

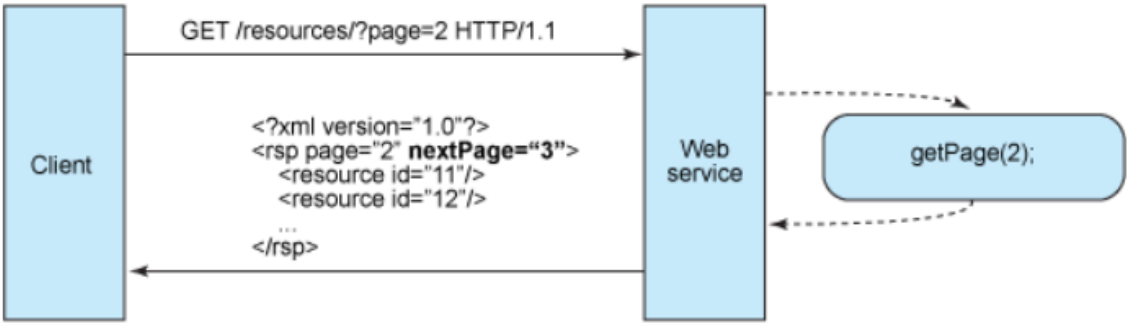
Kokonaiseen itsenäiseen pyyntöön ei vaadita kutsua palvelimelle noutamaan minkäänlaista sovelluskontekstia tai -tilaa. Luodakseen vasteen REST:n web-palvelu -sovellus (tai asiakaspää) sisällyttää pyynnön/kutsun HTTP-otsikko- ja runkotietoihin kaikki parametrit, kontekstin ja palvelinpään komponentin tarvitseman datan. Tilattomuus tässä mielessä parantaa web-palvelun suorituskykyä ja yksinkertaistaa palvelinpään komponenttien rakennetta ja toteutusta, koska tilatietojen puute poistaa tarpeen synkronoida istuntodata ulkoisen sovelluksen kanssa.

Kuva 3.4 havainnollistaa tilallisen palvelun, jossa sovellus voi pyytää seuraavan sivun usean sivun tulosjoukosta, olettaen, että palvelu seuraa, missä sovellus lopettaa joukon selaamisen. Tässä tilallisessa rakenteessa, palvelu kasvattaa `previousPage`-muuttujaa ja tallettaa sen jonnekin, jotta se pystyy reagoimaan seuraava-pyyntöihin.



Kuva 3.4 - Tilallinen rakenne (Rodriguez 2008)

Tilattomat palvelinpuolen komponentit sen sijaan ovat vähemmän monimutkaisia suunnitella, kirjoittaa sekä jakaa kuormaa tasapainottavilla palvelimilla. Tilaton palvelu ei ole ainoastaan suorituskykyisempi, se myös siirtää suurimman tilanhallinnan vastuun asiakassovellukselle. RESTful web-palvelussa palvelin on vastuussa vasteiden luomisesta ja rajapinnan tarjoamisesta, joka mahdollistaa asiakassovelluksen ylläpitää sovelluksen tilaa itse. Esimerkiksi, kun pyydetään monisivuinen tulosjoukko, asiakassovelluksen tulisi sisällyttää pyyntöön pyydettävä sivunumero, sen sijaan että pyydettäisiin vain seuraavaa sivua (kuva 3.5).



Kuva 3.5 - Tilaton rakenne (Rodriguez 2008)

Tilaton web-palvelu luo vasteen, joka linkittyy joukon seuraavaan sivunumeroon ja antaa asiakasohjelman tehdä tarvittavat toimenpiteet arvon säilyttämistä varten. Ylläpidon näkökulmasta tämän tyyppisen RESTful web-palvelun rakenne voidaan esittää alla olevan korkean tason erottelun mukaisesti:

- Palvelin
    - Luo vasteet, jotka sisältävät linkkejä toisiin resursseihin salliakseen sovelluksien navigoinnin vastaavien resurssien välillä. Tämän tyyppinen vaste upottaa linkkejä. Samoin, jos pyyntö on isäntä- tai säiliöresurssille, tyypillinen RESTful-vaste voi myös sisältää linkkejä resurssin aliresursseihin, jotta nämä pysyvät yhteydessä.
    - Luo vasteita, parantamaan suorituskykyä vähentämällä pyyntöjen määrää duplikaattiresursseille ja eliminoimalla joitain pyyntöjä kokonaan. Palve-

lin toteuttaa tämän lisäämällä Cache-Control ja Last-Modified -kentät HTTP-vasteen otsikkotietoihin.

- Asiakassovellus
    - Käyttää Cache-Control -kenttää vasteen otsikkotiedoissa, jotta voidaan selvittää siirretäänkö resurssi välimuistiin (luodaan paikallinen kopio) vai ei. Asiakassovellus lukee myös Last-Modified -kentän vasteen otsikkotiedoista ja lähettää takaisin päivämäärä-arvon If-Modified-Since -kentässä kysyäkseen palvelimelta onko resurssi muuttunut. Tätä kutsutaan ehdolliseksi GETiksi ja nämä kaksi otsikkotietoa kulkevat käsi kädessä, koska palvelimen vaste on standardi 304-koodi (Not Modified) ja jättää pois todellisen pyydetyn resurssin, jos muutoksia ei ole tapahtunut. 304 HTTP-vaste tarkoittaa, että asiakassovellus voi turvallisesti käyttää välimuistissa olevaa paikallista kopiota resurssista, samalla ohittaen myöhempiä GET-pyyntöjä siihen asti, kunnes resurssi muuttuu.
    - Lähettää kokonaisia pyyntöjä, joita voidaan palvella itsenäisesti muista pyynnöistä riippumatta. Tämä vaatii, että asiakasohjelma hyödyntää täysmääräisesti HTTP-otsikkotietoja niin kuin on määritelty web-palvelun rajapinnassa ja lähettää kokonaisia kuvauksia (representations) resursseista pyynnön rungossa. Asiakasohjelma lähettää pyyntöjä, jotka tekevät erittäin vähän olettamuksia aiemmista pyynnöistä, istunnon olemassa olosta palvelimella, palvelimen kyvystä lisätä konteksti pyyntöön, tai sovelluksen tilasta, joka pidetään pyyntöjen välissä.

Yhteistyö asiakassovelluksen ja palvelun välillä on olennaista tilattomuudelle RESTful web-palvelussa. Tilattomuus parantaa tehokkuutta säästämällä kaistaa ja minimoimalla palvelinpuolen sovellustilan ylläpidon.

Hakemistokaltainen URI-rakenne

REST web-palvelun URI:n tulisi olla niin intuitiivisia, että ne olisi helppo arvata. URI:a voisi ajatella eräänlaisena itsedokumentoituvana rajapintana, joka vaatii hyvin vähän, jos ollenkaan, selityksiä tai viitteitä, jotta kehittäjä ymmärtää mihin se osoittaa ja miten sieltä saa siihen liittyviä resursseja. Tästä syystä URI:n rakenteen tulisi olla suoraviivainen, ennustettava ja helposti ymmärrettävä.

Yksi tapa saavuttaa tämän tason käytettävyyttä on määritellä URI:t hakemistorakenteen mukaisiksi. Tämän tyyppinen URI on hierarkkinen: juuri-solmuun on yksi polku ja siitä haarautuvat ovat alipolkuja, jotka näyttävät palvelun pääalueet. Tämän määritelmän mukaan URI ei ole ainoastaan kauttaviivoilla rajattu merkkijono vaan puu, jolla on ylemmän ja alemman tason lehtiä solmukohdilla yhdistettynä. Esimerkiksi puheenaiheita keräävä keskustelupalstan URI-joukko voitaisiin määritellä seuraavasti:

```
http://www.myservice.org/discussion/topics/{topic}
```

Juurena toimii `/discussion` -solmu ja sen alapuolella on `/topics` -solmu, jonka alapuolella on joukko puheenaiheiden nimiä (`topics`), jotka puolestaan osoittavat keskusteluketjuun (`discussion`). Tämänlaisessa rakenteessa on yksinkertaista hakea keskusteluketjuja vain kirjoittamalla jotain `/topics/` -solmun jälkeen.

Joissakin tapauksissa polku resurssiin soveltuu erityisen hyvin hakemistorakenteen kaltaiseen rakenteeseen. Esimerkiksi resurssit, jotka ovat järjestetty päivämäärän mukaan sopivat hyvin hierarkkiseen syntaksiin.

```
http://www.myservice.org/discussion/2008/12/10/{topic}
```

Ensimmäinen polun osa on nelinumeroinen vuosi, toinen polun osa on kahden numeron kuukausi ja kolmas polun osa on kaksinumeroinen päivä. Ihmisten ja koneiden on helppo luoda tämän tyyppisiä rakenteellisia URI:a, koska ne perustuvat sääntöihin.

```
http://www.myservice.org/discussion/{year}/{month}/{day}/{topic}
```

Muutamia suuntaviivoja tulisi ottaa huomioon suunnitellessa URI-rakennetta RESTful web-palvelua varten:

- Palvelinpään skriptausteknologian tiedostopäätteet tulisi piilottaa (.jsp. .php, .asp), jotta teknologian vaihtaminen onnistuu ilman URIn muuttamista.
- Kaikki tulisi olla pienellä kirjaimilla kirjoitettu (lower case)
- Välilyönnit tulisi korvata joko tavuviivoilla tai alaviivoilla.
- Kyselymerkkijonoja tulisi välttää.
- Vasteena tulisi tarjota oletussivu tai -resurssi, sen sijaan, että käytettäisiin 404 Not Found -virhekoodia, jos pyydetty URI on osittaiseen polkuun.

URIn tulisi olla staattinen, koska resurssin muuttuessa tai palvelun toteutuksen muuttuessa, linkki pysyisi samana. Tämä esimerkiksi ylläpitää kirjanmerkkien toimivuutta.

XML ja JSON -tiedonsiirto

Resurssin esitystapa tyypillisesti heijastaa resurssin nykyistä tilaa ja sen attribuutteja sillä ajanhetkellä, kun asiakassovellus sitä pyytää. Tässä mielessä resurssien esitykset ovat vain tilannevedoksia. Tämä voi olla niinkin yksinkertainen kuin tietokannassa oleva tietue, joka koostuu sarakkeiden ja XML-tunnisteiden kartoituksesta, jossa XML:ssä olevien elementtien arvot sisältäisivät rivien arvot. Tai, jos järjestelmällä on tietomalli, resurssiesitys on järjestelmän attribuuttien tilannevedos.

RESTful web-palvelun rakenteen viimeinen rajoite liittyy datan formaattiin, jolla sovellus ja palvelu vaihtavat pyynnön ja vasteen hyötykuormaa, tai HTTP-runkoon. Tällä saadaan pidettyä asiat yksinkertaisina, ihmisenluettavana ja yhtenäisenä.

Tietomallin objektit liittyvät yleensä toisiinsa jollakin lailla, ja suhteet tietomallin resurssien välillä tulisi havainnollistaa siten kuin ne näyttävät siirrettäessä asiakassovellukselle. Keskustelupalstapalvelussa esimerkki yhtenäisistä resurssiesityksistä voisi pitää sisällään keskusteluaiheen juuren ja sen attribuutit, ja upottaa linkkejä siihen aiheeseen liittyviin vasteisiin (listaus 3.6).

Listaus 3.6 - Keskusteluketjun XML-esitys

```
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
   <comment>{comment}</comment>
   <replies>
      <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
      <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
   </replies>
</discussion>
```

Ja lopuksi, asiakassovelluksille tulisi antaa kyky pyytää tiettyä sisältötyyppiä, joka niille parhaiten sopii. Palvelu tulisi rakentaa niin, että se käyttäisi sisäänrakennettua HTTP Accept -otsikkotietoja, jossa otsikkotiedon arvo on MIME-tyyppi. Taulukossa 3.1 on esitelty RESTful-palveluiden yleisimmin käytetyt MIME-tyypit.

Taulukko 3.1 - RESTful-palvelun yleisimmin käytetyt MIME-tyypit (Rodriguez 2008)

MIME-tyyppi	Sisältötyyppi
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

Tämä mahdollistaa palvelun käytön useilla asiakassovelluksilla, jotka on kirjoitettu eri kielillä ja toimivat eri alustoilla sekä laitteilla. Kun käytetään MIME-tyyppejä sekä HTTP Accept -otsikkotietoja, se tunnetaan sisältöneuvottelumekanismina, joka sallii asiakassovellusten valita niille sopivan dataformaatin ja minimoi datakytkennät (data coupling) palvelun ja sovellusten välillä.

Yhteenveto

REST ei ole aina oikea ratkaisu. Se on saanut kehittäjien kiinnostuksen, koska sillä voidaan suunnitella web-palvelut, jotka ovat vähemmän riippuvaisia väliohjelmistoista (middlewaresta), kuin SOAP ja WSDL -tyyppiset ratkaisut. Tietyllä tavalla REST on paluu aikaan ennen isoja sovelluspalvelimia varhaisen Internetin standardien, URI:n ja

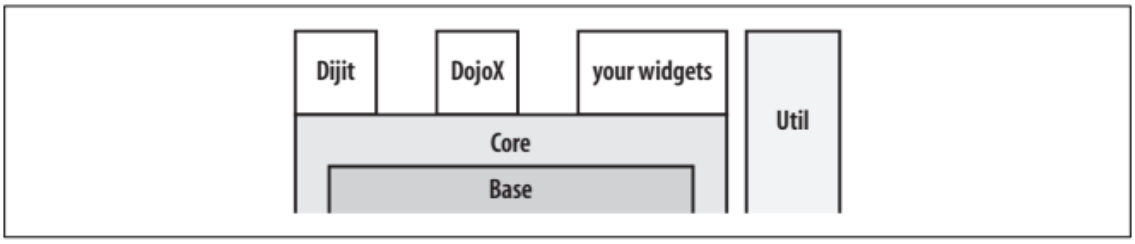
HTTP:n, kautta. HTTP:n yli käytettävä XML on rajapinta, joka sallii sisäisten sovellusten, kuten Asynchronous JavaScript + XML (Ajax)-pohjaisten räätälöityjen käyttöliittymien, yhdistää, osoittaa ja kuluttaa resursseja. Järjestelmän resurssien paljastaminen RESTful-API:n kautta on joustava tapa toimittaa erilaisia sovelluksia standardia noudattavan muotoillun datan kanssa.

3.3 Selainpään työkalupakki

Tässä kohdassa esitellään Dojo Toolkit, joka on nimensä mukaisesti työkalupakki selainsovelluksien toteutukselle JavaScript-kielellä. Lisäksi esitellään pintapuolisesti JavaScript-kieltä ja suoritetaan kevyttä vertailua Javan ja JavaScriptin välillä.

Dojo Toolkit

Dojo Toolkit tarjoaa JavaScript kirjaston, jossa on mukana muiden muassa ominaisuusrikkaita pienoisohjelmia (widget) ja käännöstyökaluja. Kirjasto sisältää myös monipuolisen testausalustan. Kuvassa 3.6 on esitetty Dojon arkkitehtuuri. DojoX on esitetty itsenäisenä kokonaisuutena, DojoX-resurssit voisivat rakentua esimerkiksi Dijitin resurssien varaan, samalla tavalla kuten itse rakennetut pienoisohjelmat voivat hyötyä mistä tahansa yhdistelmästä Dijit ja DojoX -resursseja.



Kuva 3.6 - Kuvaus siitä, miten eri Dojon komponentit voidaan ajatella olevan liitettynä toisiinsa (Russell 2008)

Tässä kohdassa on käytetty lähteenä Russellin (2008) kirjoittamaa opasta Dojo Toolkitin ominaisuuksista.

Base

Dojon ydin Base on tiivis ja optimoitu kirjasto, joka tarjoaa perustan kaikelle muulle työkalupakissa. Base tarjoaa muiden muassa kieli ja Ajax -apuohjelmat, paketointijärjestelmän, joka mahdollistaa Dojon resurssien hakemisen lennosta, sen sijaan, että ne tarvitsisi kaikki ladata kerralla sivun latautuessa. Sen mukana tulee myös työkalut periytymishierarkioiden luomiseen ja manipulointiin, välineet lähes universaalille DOM ja CSS -valitsimien kyselyihin, ja rakenne, joka standardoi DOM-tapahtumat eri selaimien keskuudessa. Kaikki mitä Base tarjoaa on saatavilla työkalupakin nimiavaruuden ylimmältä tasolta dojo.* -funktiona tai -attribuuttina. Base tulee dojo.js-tiedostoon pakattuna.

Base on suunniteltu esilataamaan Dojon keskeiset osat automaattisesti, kun dojo.js on asetettu sivuun. Esilataaminen tuo mukanaan ympäristön tunnistamisen, selainyhteensopimattomuuksien tunnistamiset ja sovittamiset sekä Dojon nimiavaruuden lataamisen. Useita asetuksia voidaan määrittää, jotta sivulla olevat pienoisohjelmat voidaan automaattisesti parsia, sekä suorittaa muita käynnistymiseen liittyviä tehtäviä.

Base tarjoaa apuohjelmia useisiin vakiotoimenpiteisiin, joita yleensä tarvitaan JavaScriptin parissa työskentelyyn. Ei ole Dojoa ilman Basea; kaikki työkalupakissa olevat ovat riippuvaisia tai rakentuvat siihen tavalla tai toisella.

Core (dojo)

Core, jota kutsutaan nykyään ennemmin dojoksi, rakentuu Basen päälle tarjoamalla lisäpalveluja kuten pienoisohjelmien parsiminen, kehittyneet animaatiotehosteet, vedä ja pudota, kansainvälistäminen (i18n), takaisin-napin hallinta, sekä evästeiden hallinnan. Coresta saatavia resursseja käytetään usein tarjoamaan tukea tavallisille operaatioille. Dojon paketointijärjestelmä ansiosta lisämoduulien käyttöönottaminen on mahdollistettu yksinkertaisella mekanismilla, joka toimii samalla periaatteella kuin Javan import-lauseke.

Yleisesti Basen ja Coren erottaminen toisistaan on yksinkertaista: mikä tahansa moduuli tai resurssi, joka tarvitsee erikseen tuoda sivuun dojo.js:n ulkopuolelta, on osa Corea, jos se kuuluu Dojon nimiavaruuteen. Coren palvelut eivät yleensä esiinny Basen nimiavaruuden tasolla, vaan esiintyvät alemman tason nimiavaruudessa, kuten dojo.fx tai dojo.data.

Dijit

Dojossa on myös mukana pienoisohjelmien kirjasto, jota kutsutaan Dijitiksi (lyhenne sanoista Dojo Widget), joka on heti valmis käytettäväksi ja usein se ei tarvitse yhtään kirjoitettua JavaScript-koodia. Dijit on rakennettu suoraan Coren päälle, joten aina, kun tulee tarve määritellä oma pienoisohjelma, käytetään samoja rakennuspalikoita kuin käytettiin rakentamaan kaikki muukin Dijitissä. Kaikki Dojolla itse rakennetut pienoisohjelmat ovat siirrettäviä ja voidaan helposti jakaa tai ottaa käyttöön web-palvelimella tai ajaa paikallisesti ilman web-palvelinta file:// -syntaksin mukaisesti.

Dijitin liittäminen sivuun onnistuu yksinkertaisimmillaan määrittelemällä Dojon tyyppitunnisteet HTML-tunnisteiden sisälle. Tärkein hyöty Dijitin käytöstä sovelluskehittäjille on se, että sillä pystytään saavuttamaan monipuolisia toiminnallisuuksia ilman, että täytyisi syventyä toteutuksen yksityiskohtiin. Vaikka suosittaisiin kirjastokirjoittamistyyliä tai räätälöityjä pienoisohjelmia, seuraamalla Dijitin tyyliä ja yleisiä

tapoja, voidaan varmistaa pienoisohjelmien siirrettävyys ja helppokäyttöisyys, joka on välttämätöntä mille tahansa uudelleenkäytettävälle komponentille.

Dijitin sisältö voidaan karkeasti jakaa yleiskäyttöisiin sovelluspienoisohjelmiin, kuten edistymispalkkeihin ja modaalidialogeihin, ulkoasupienoisohjelmiin kuten välilehtisäiliöihin ja haitariruutuihin, sekä pienoisohjelmiin, jotka tarjoavat kehittyneempiä versioita napeista ja muista syöte-elementeistä.

DojoX

DojoX on kokoelma aliprojekteja (Dojo Extensions), jotka eivät ole sopineet Coren (Dojon) tai Dijitin joukkoon. DojoX:n kokeellisissa aliprojekteissa on mukana pienoisohjelmia, jotka ovat epävakaita ja kehityksen alla. Jokaisessa DojoX:n aliprojektissa on oltava mukana README-tiedosto, joka sisältää yleiskatsauksen sen tilasta. DojoX tarjoaa hiekkalaatikon ja hautomon uusille ideoille, samalla kun pidetään Core ja Dijit koskemattomina.

Util

Util on kokoelma Dojo-apuohjelmia, jotka sisältävät JavaScript yksikkötestausalustan sekä käännöstyökaluja luomaan räätälöityjä versioita Dojosta tuotantoympäristöön. Esimerkiksi yksikkötestausalustalla, DOH, ei ole mitään erityistä kytkentää Dojoon ja tarjoaa yksinkertaisen joukon rakennelmia, joilla voidaan automatisoida laadunvarmistusta mille tahansa JavaScript koodille.

JavaScript

JavaScriptin standardoidu version on nimeltään ECMAScript. Yritykset voivat käyttää vapaata standardikieltä rakentaakseen omat JavaScript-versionsa. ECMAScriptin standardi on dokumentoitu Ecma International ECMA-262 (2011) -spesifikaation painokseen 5.1. Tässä alakohdassa on käytetty lähteenä Mozillan kehittäjäverkoston (MDN 2014) laatimaa JavaScriptin yleiskatsausta.

JavaScript on monen alustan olio-pohjainen skriptikieli. JavaScript on pieni ja kevyt kieli; se ei ole kovin hyödyllinen yksinään (standalone), mutta se on suunniteltu helposti sisällytettäväksi muihin tuotteisiin ja sovelluksiin, kuten web-selaimiin. JavaScript voidaan yhdistää toimintaympäristönsä objekteihin isäntäympäristönsä sisällä, jotta niihin saataisiin ohjelmallinen kontrolli.

JavaScriptin ydin sisältää joukon ydinobjekteja (core objects), kuten Array, Date, ja Math, sekä kielen elementtien ydinjoukon, kuten operaattorit, kontrollirakenteet, ja lausekkeet. JavaScriptin ydintä voidaan laajentaa useaa tarkoitusta varten täydentämällä

sitä ylimääräisillä objekteilla. Laajennukset voidaan jakaa selainpään ja palvelinpään laajennuksiin.

Selainpään (client-side) JavaScript laajentaa kielen ydintä tarjoamalla objekteja hallitsemaan selainta sekä sen DOM:a (Document Object Model). Esimerkiksi, selainpään laajennukset sallivat sovelluksen asettaa elementtejä HTML-sivulle sekä vastata käyttäjätapahtumiin, kuten hiiren klikkauksiin, lomakkeiden syötteisiin ja sivunavigointiin.

Palvelinpään (server-side) JavaScript laajentaa kielen ydintä tarjoamalla objekteja, jotka ovat olennaisia JavaScriptin ajamiseen palvelimella. Esimerkiksi, palvelinpään laajennukset sallivat sovelluksen ottaa yhteyden relaatiotietokantaan, tarjota informaation pysyvyyttä sovelluksen pyynnöltä toiselle, tai suorittaa tiedostomanipulaatiota palvelimella. Java ja JavaScript koodit voidaan sallia kommunikoida keskenään JavaScriptin LiveConnect -toiminnallisuuden kautta. JavaScriptin puolelta voidaan luoda (instantiate) Java-objekteja, sekä päästään käsiksi julkisiin metodeihin ja kenttiin. Javan puolelta saadaan pääsy JavaScript-objekteihin, ominaisuuksiin ja metodeihin.

JavaScript vs. Java

JavaScript ja Java ovat samantyyppisiä jollakin lailla, mutta perustavanlaatuisesti erilaisia toisilla tavoilla (taulukko 3.2). JavaScriptin kieli muistuttaa Javaa, mutta sillä ei ole Javan staattista tyypitystä ja vahvaa tyyppitarkistusta. JavaScript seuraa suurinta osaa Javan lausesyntaksista, nimeämiskäytännöistä ja perus kontrollivuorakenteista, mistä syystä se nimettiin uudelleen LiveScriptistä JavaScriptiksi.

Taulukko 3.2 JavaScript verrattuna Javaan. (MDN 2014)

JavaScript	Java
Olio-orientointunut. Ei erottelua eri oliotyyppien välillä. Perintä tapahtuu prototyyppimekanismilla, sekä ominaisuudet ja metodit voidaan lisätä mihin tahansa olioon dynaamisesti.	Luokka-pohjainen. Oliot jaetaan luokkiin ja ilmentymiin, joihin kaikki perintä tapahtuu luokkahierarkian kautta. Luokkiin ja ilmentymiin ei voi lisätä ominaisuuksia metodeja dynaamisesti.
Muuttujan tietotyyppejä ei määritellä (dynaaminen tyypitys).	Muuttujan tietotyypit täytyy määritellä (staattinen tyypitys).

JavaScript tukee ajonaikaista järjestelmää, joka perustuu pieneen määrään datatyyppejä, jotka edustavat numeerisia-, Boolean-, ja merkkijonoarvoja, toisin kuin Javan käännettäessä esittelyillä rakennettu luokkajärjestelmä. JavaScriptillä on prototyyppipohjainen olioiden mallinnus, paljon yleisemmän luokkapohjaisen oliomallin sijasta. Prototyyppipohjainen malli tarjoaa dynaamisen perinnän. Toisin sanoen, periytymisen sisältö riip-

puu yksittäisistä olioista. JavaScript tukee myös funktioita ilman erityisiä esittelyvaatimuksia.

JavaScript on erittäin vapaamuotoinen kieli verrattuna Javaan. Kaikkia muuttujia, luokkia tai metodeja ei tarvitse esitellä. Ei tarvitse huolehtia, mitkä metodit ovat julkisia, suojattuja tai salaisia, eikä rajapintoja tarvitse toteuttaa. Muuttujia, parametreja ja metodien palautustyyppejä ei ole eksplisiittisesti tyypitetty.

Java on luokkapohjainen ohjelmointikieli, joka on suunniteltu nopeaan suoritukseen ja tyyppiturvallisuuteen. Tyyppiturvallisuus tarkoittaa esimerkiksi sitä, että Javassa kokonaislukutyyppistä oliota ei voida muuttaa olioviitteeksi tai voida saada pääsy yksityiseen muistiosioon korruptoimalla Javan tavukoodit. Javan luokkapohjainen malli tarkoittaa sitä, että ohjelmat koostuvat yksinomaan luokista ja niiden metodeista. Javan luokkaperiytyminen ja vahva tyypitys yleisesti vaativat tiukasti kytketyt oliohierarkiat. Nämä vaatimukset tekevät Java-ohjelmoinnista monimutkaisempaa kuin JavaScript-ohjelmointi.

JavaScript puolestaan polveutuu pienempien, dynaamisesti tyypitettyjen kielien linjasta, kuten HyperTalk ja dBASE. Nämä skriptikielet tarjoavat ohjelmointityökaluja laajemmalle yleisölle helpomman syntaksin, erityisen sisäänrakennetun toiminnallisuuden, ja olionluonnin pienten vaatimusten ansiosta.

4 TULOSKONSTRUKTIO

Tässä luvussa esitellään aikaan saatu elektronisen prosessioppaan sovellus, joka esitellään neljässä aliluvussa. Ensimmäisessä kohdassa esitellään sovelluksen käyttötapaukset ja arkkitehtuuri, joiden tarkoituksena olisi antaa määrittelytason kuvaus prosessioppaan kaikista osista. Toisessa kohdassa esitellään sovelluksen tietokantaosuus. Kolmannessa kohdassa esitellään tutkimuksen nimessäkin luvattu koneluettava rajapinta ja komponentit. Viimeisessä kohdassa esitellään prosessiopassovelluksen käyttäjälle näkyvä-osa eli sen käyttöliittymä.

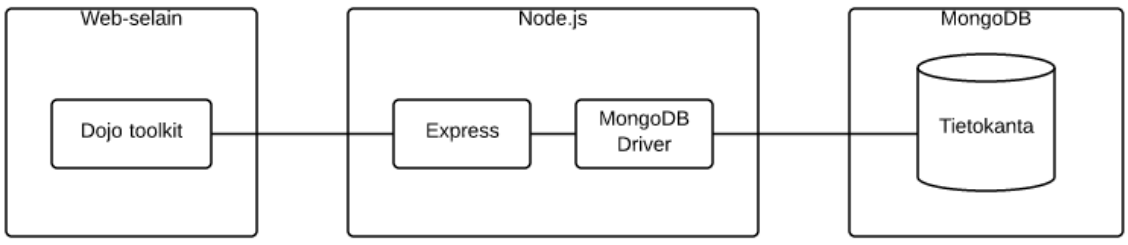
4.1 Käyttötapaukset ja arkkitehtuuri

Tässä kohdassa esitellään elektronisen prosessioppaan määrittelytason kuvaus sovelluksen kaikista osa-alueista.

Yleiskuvaus

Web-sovelluksen on tarkoitus olla ISO/IEC TR 29110-5-1-1:2012(E) -standardiin pohjautuva elektroninen prosessiopas. Oppaaseen on koottu standardin prosessit, prosessien aktiviteetit, ja aktiviteettien tehtävät, sekä niiden kuvaukset. Lisäksi oppaasta löytyy roolien ja tuotteiden kuvaukset.

ISO/IEC TR 29110 -standardin analysoinnin pohjana on käytetty Mäkisen ja Varkoin (2008) julkaisua. Elektronisen prosessioppaan tapauksessa käyttäjälle näkyvän prosessioppaan pohjana on käytetty Alexandre et al. (2008) julkaisua.



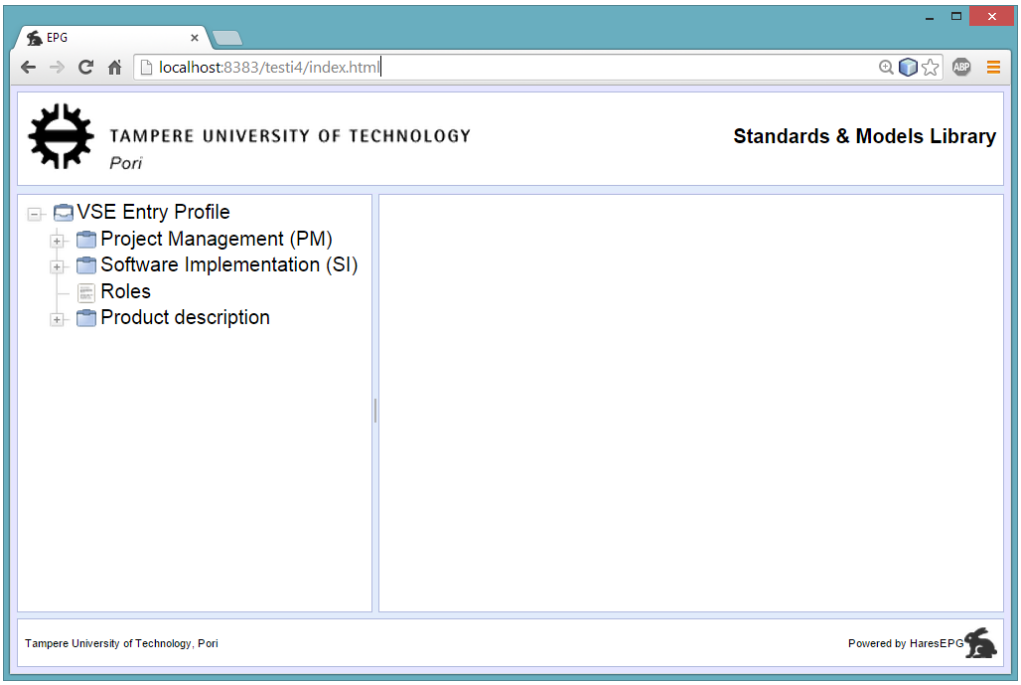
Kuva 4.1 - Teknisen ratkaisun rakenne

Kuvassa 4.1 on esitetty elektronisen prosessioppaan rakenne. Kuvassa 4.2 näkyvä käyttöliittymä on toteutettu JavaScript-pohjaista Dojo-työkalupakkia käyttämällä. Node.js -sovellus (jäljempänä Node) toteuttaa REST-rajapintatoiminnallisuutta ladattavien kolmansien osapuolien avulla sekä sen data sijaitsee JSON-muotoisena MongoDB-tietokantapalvelimella. Dojo toolkit lähettää HTTP GET -kutsuja Noden Express-

moduulille, jossa kutsut muutetaan dokumenttienhakukutsuiksi Noden MongoDB-ajurin avulla MongoDB:en. Web-sovelluksen rakentamisen pohjana on käytetty useita eri oppaita ja dokumentaatioita Dojo toolkitin web-sivuilta. REST-toiminnallisuuden rakentamisen oppaana on käytetty Coenraetsin (2012) laatimaa opasta. MongoDB:en on talletettu JSON-muotoisena datana ISO/IEC TR 29110-5-1-1:2012(E) -standardin osittainen sisältö.

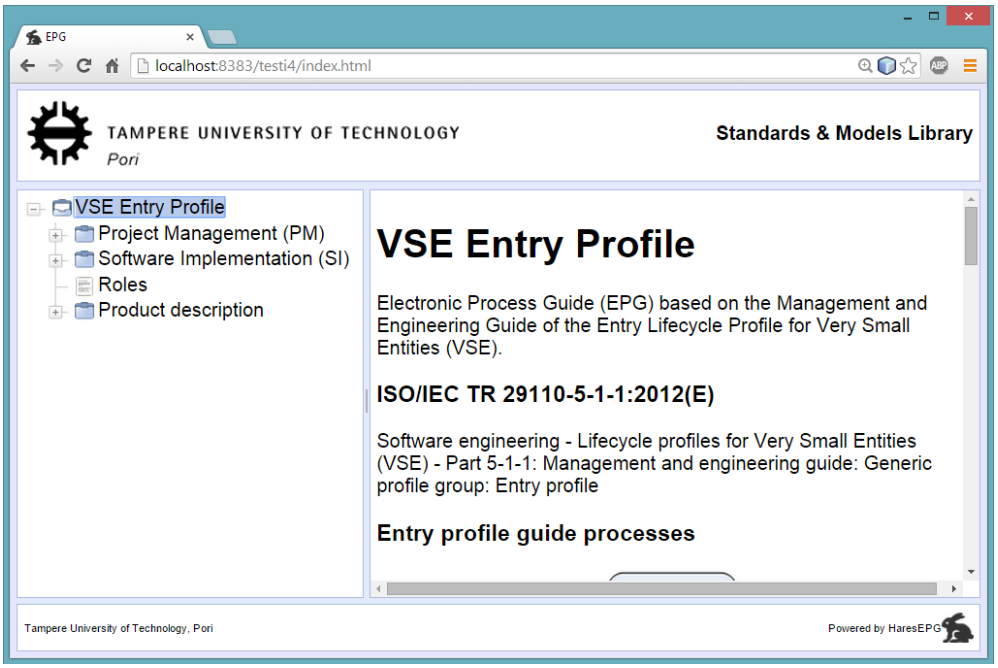
Ulkoasu ja ulkoinen käyttäytyminen

Sivun latauduttua selain-ikkuna näyttää kuvan 4.2 mukaiselta. Selain-ikkuna on jaettu neljään paneeliin, joista ylä- ja alapaneelin sisällöt on säädetty staattisiksi eikä niiden sisältö muutu käytön aikana.



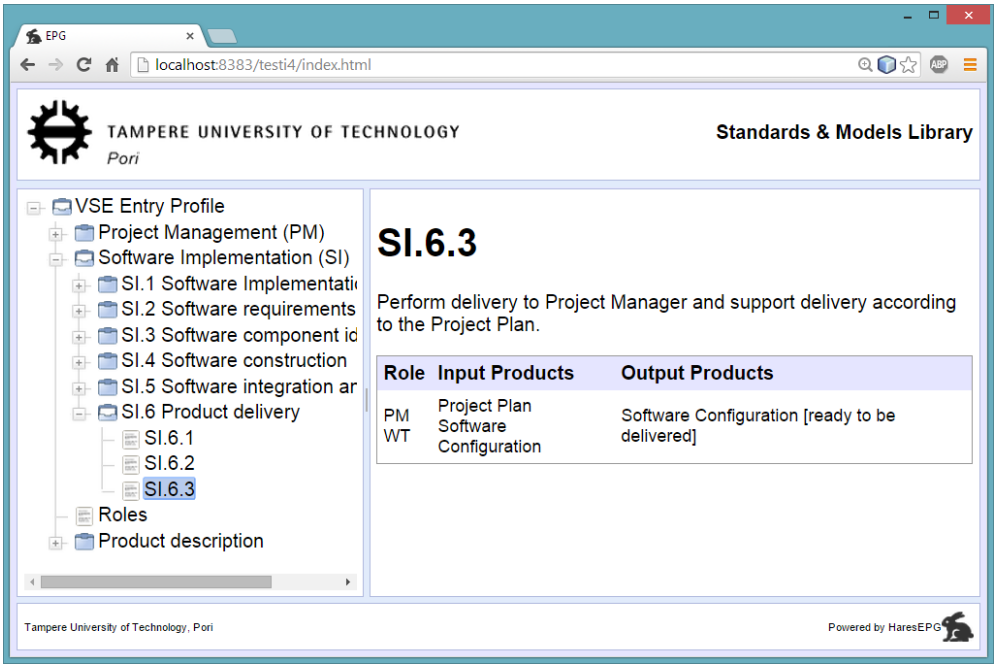
Kuva 4.2 - Web-sovelluksen ulkoasu web-selaimessa

Selain-ikkunan vasemman puoleisessa keskipaneelissa on puu-käyttöliittymäelementti, jonka lapsielementtejä käyttäjä voi valita klikkaamalla. Kun puun jotakin elementtiä on klikattu, ikkunan keskipaneeliin (center, oikean puoleinen) ilmestyy puun klikattuun elementtiin liittyvää tietoa kuvan 4.3 mukaisesti.



Kuva 4.3 - Puun elementti valittuna

Käyttäjä voi myös avata puun solmuja, jolloin ladataan lisää dataa rajapinna kautta ja Dojo yhdistää sen käyttöliittymän elementtiin, ja klikata niiden lapsielementtejä (Kuva 4.4).



Kuva 4.4 - Puun solmuja avattuna ja elementti valittuna

Käyttäjän vuorovaikutus

Käyttäjän näkökulmasta web-sovelluksen käytössä on ainoastaan kaksi käyttötapausta, koska käyttöliittymässä on pyritty yksinkertaisuuteen. Käyttäjä klikkaa puun elementtiä ja käyttäjä avaa puun solmun. Kun selain-ikkuna aukeaa, siihen latautuu kuvan 4.2 mu-

kainen käyttöliittymä, johon puu-elementti on latautunut käyttöliittymän keskellä vasempaan (leading) paneeliin.

Käyttäjä klikkaa puun elementtiä

Kun käyttäjä klikkaa puussa olevaa elementtiä, näytetään klikattuun elementtiin liittyvää tietoa selain-ikkunan keskipaneelissa. Tässä tapauksessa web-sovelluksen ei tarvitse ladata dataa muualta lisää, kun kaikki näkyvät elementit on jo ladattu asiakassovelluksen käynnistymisvaiheessa.

Käyttäjä avaa puun solmun

Kun käyttäjä avaa puun solmun, web-sovelluksen tarvitsee ladata lisää JSON-muotoista dataa. Kuvassa 4.4 on avattu puunsolmuja. Klikattavassa elementissä oleva `_id`-avain määrittää sen, mitä dokumenttia pyydetään rajapinnalta.

Esimerkiksi puuhun on selain-ikkunan auetessa REST-arkkitehtuurityylin mukaisesti latautunut vse-tree -dokumentti (dokumentin juuressa on `"_id"="vse-tree"` avain/arvo-pari), jonka sisältöä esitellään myöhemmässä luvussa.

Käyttäjä avaa puuelementistä esimerkiksi Product Descriptions -solmun, jolloin käyttöliittymä pyytää REST-rajapinnalta Product Descriptions -elementtiin liittyvät lapsielementit. Rajapinta palauttaa pyydetyt elementit JSON-muotoisena, jonka jälkeen käyttöliittymä lataa ne puuhun ja uusi data tulee käyttäjän saataville.

Käynnistyminen

Elektroninen prosessiopas koostuu kolmesta erillisestä osasta: MongoDB-tietokannasta, Node-palvelimesta ja web-sovelluksesta (kuva 4.1). Käynnistysjärjestyksessä ensimmäisenä on MongoDB, joka saattaa dokumentit saataville. Seuraavaksi käynnistetään Node-palvelin, joka toteuttaa koneluettavan rajapinnan dokumenttien noutamiselle REST-arkkitehtuurityylin mukaisesti. Viimeisenä käynnistetään web-sovellus, jonka käyttöliittymä mukautuu rajapinnan kautta noudettavaan JSON-muotoiseen dataan.

4.2 Tietokanta

Tässä kohdassa esitellään miten elektronisen prosessioppaan sisältö on talletettu MongoDB-dokumenttitietokantaan.

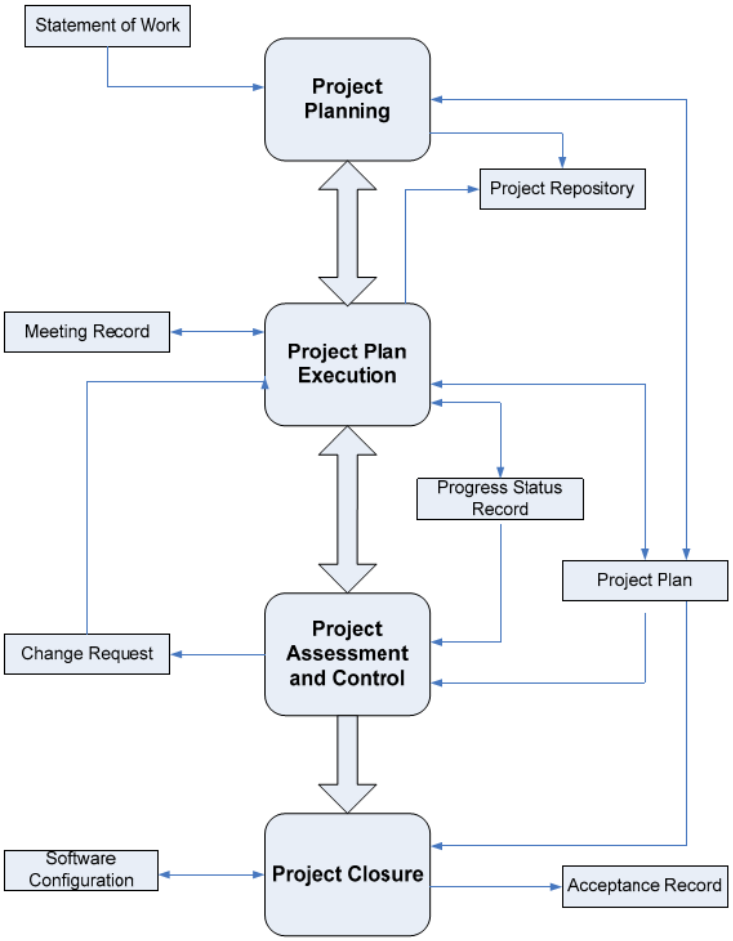
ISO/IEC TR 29110-5-1-1 sisältö JSON-dokumenteiksi

MongoDB:n talletetut JSON-dokumentit perustuvat standardiin ISO/IEC TR 29110-5-1-1:2012(E), jossa on määritelty ohjelmistotuotannon elinkaariprofiilit pienille ohjelmistoyksiköille. Standardin osa 5-1-1 on Entry-tason hallinnan, suunnittelun ja toteutuksen opas.

Koska sovelluksen on tarkoitus olla elektroninen prosessiopas ISO/IEC TR 29110-5-1-1:2012(E) standardista, standardin sisältö on tarvinnut analysoida. Tähän on käytetty pohjana Mäkisen ja Varkoin (2008) julkaisua, jossa analysoidaan ISO/IEC TR 29110 Basic profiilin aikaisen luonnoksen sisältöä. Lähteenä on myös käytetty Alexandren et al. (2008) julkaisua, jossa on tarkoitus sovittaa edellisessä julkaisussa analysoitu standardi elektronisen prosessioppaan muotoon käyttämällä Eclipse Process Framework -työkalua.

ISO/IEC TR 29110-5-1-1:2012(E):stä voidaan tunnistaa kaksi prosessia: projektinhallinta (Project Management, PM) ja ohjelmiston toteutus (Software Implementation, SI). Prosessien tunnistaminen tapahtuu muiden muassa Mäkisen ja Varkoin (2008) julkaisuun perehtymällä, sekä ISO/IEC TR 29110-5-1-1:2012(E):n sisällysluetteloa ja sen hierarkiaa tutkimalla, sekä standardista löytyvillä prosessien kaavioilla. Kuvissa 4.5 ja 4.6 on esitelty kummankin prosessin aktiviteetit ja niihin liittyvät tulo- ja lähtötuotteet.

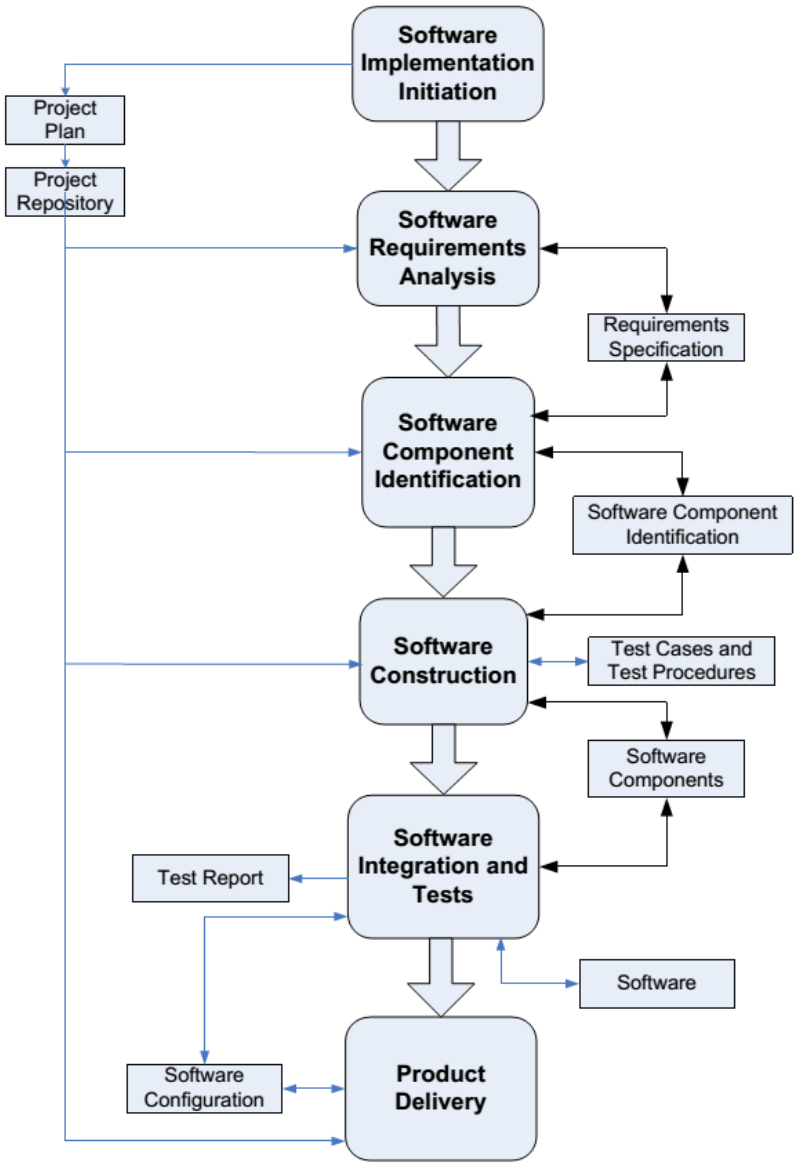
Projektinhallintaprosessi (PM) on standardin luvun 6 otsikkona ja sen aliluvussa on esitelty aktiviteetit, joita PM-prosessin tapauksessa on neljä. Prosessin hierarkiassa syvemmällä PM-prosessin aktiviteetin alla on aktiviteettiin liittyvät tehtävät. Tehtävien yhteyteen on muiden muassa määritelty niihin liittyvät roolit ja tulo-, sisäiset- ja lähtötuotteet.



Kuva 4.5 - Projektinhallintaprosessin kaaviokuva (ISO/IEC TR 29110-5-1-1:2012(E))

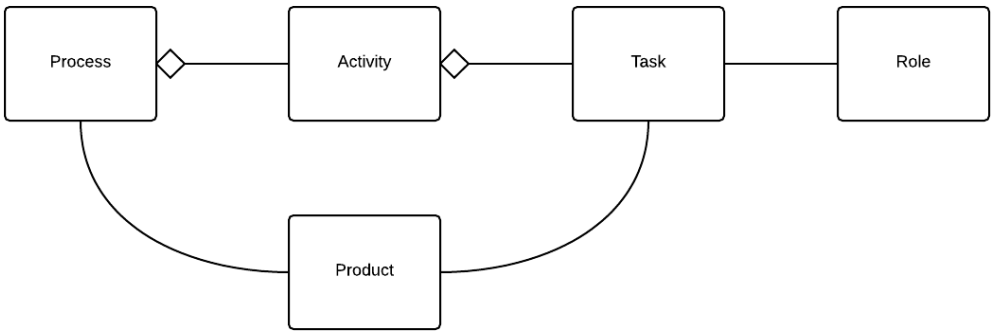
Näistä tiedoista voidaan hahmotella kaavio standardin sisällöstä, joka toimii rakennemallina JSON-dokumenteille. Kuvassa 4.7 on esitetty tietosisällön rakennetta, eli on tunnistettu prosesseja, aktiviteetteja, tehtäviä, rooleja sekä tuotteita. Kuvassa 4.7 on esitetty, että prosessit koostuvat aktiviteeteista ja aktiviteetit koostuvat tehtävistä. Yksi tai useampi rooli on vastuussa tehtävästä. Prosessilla on tulo-, sisäisiä- ja lähtötuotteita. Lisäksi tehtävillä voi olla tulo- ja lähtötuotteita ja tästä saadaan kuvassa 4.7 näkyvä tehtävän ja tuotteen välinen yhteys.

Lisäksi määrittely voi jäädä korkeammalle tasolle JSON-dokumenttien kanssa, kun esimerkiksi relaatiomallissa tarvitsisi selvittää perin pohjin esimerkiksi prosessien ja tuotteiden suhteet. JSON-dokumentin tapauksessa tieto voidaan järjestää intuitiivisesti sisäkkäin. Esimerkiksi ISO/IEC TR 29110-5-1-1:2012(E) -dokumentissa prosessien alta löytyvät aktiviteetit, joten JSON-dokumentti voidaan rakentaa ECMA-404 spesifikaation (2013) mukaisesti siten, että aktiviteetit ovat prosessin lapsielementtejä (listaus 4.1). Dokumenttimallin kääntöpuolena on, että samaa tietoa joudutaan toistamaan, kun esimerkiksi relaatiomallissa voidaan niin sanotusti viitata tarvittavaan tietoon.



Kuva 4.6 - Ohjelmiston toteutus -prosessin kaaviokuva (ISO/IEC TR 29110-5-1-1:2012(E))

ISO/IEC TR 29110-5-1-1:2012(E):n dokumenttia analysoimalla päädyttiin kuvan 4.7 mukaiseen kaavioon, joka havainnollistaa elektronisen prosessioppaan tietolähteenä toimivien JSON-dokumenttien rakennetta. JSON-dokumentteihin kootaan siis ISO/IEC TR 29110-5-1-1:2012(E):n sisältämä data, joka säilötään MongoDB-tietokantaan.



Kuva 4.7 - Kaavio standardin pääkohdista

ECMA-404 spesifikaation (2013) mukaan JSON-dokumentin sisään voidaan sisällyttää toinen dokumentti. Dokumenteista on rakennettu puun muotoisia, ja näin ollen sisältävät toisia dokumentteja. Aktiviteetit voidaan asettaa JSON-dokumentissa prosessin lapsielementeiksi. Listaus 4.1 havainnollistaa tätä esimerkiksi siten, että Activities-objekti on asetettu Project Management (PM):n lapsielementiksi. JSON-dokumenteista on tehty kahden tason syvyisiä, jolloin JSON-dokumentteja ei tarvita niin montaa ja yksi MongoDB:sta ladattu JSON-dokumentti sisältää enemmän dataa.

Listaus 4.1 - vse-tree -dokumentin rakennetta havainnollistava koodilistaus

```
{
    "version" : "2014-02-11",
    "_id" : "vse-tree",
    "name" : "VSE Entry Profile",
    "description" : "Electronic Process Guide (EPG) based on the Management
and Engineering Guide of the Entry Lifecycle Profile for Very Small Entities
(VSE).",
    "items" : [
        {
            "name" : "ISO/IEC TR 29110-5-1-1:2012(E)",
            "description" : "..."
        },
        {
            "image" : "./images/entry-profile-guide-processes.png",
            "name" : "Entry profile guide processes"
        },
        {
            "name" : "Target Group",
            "description" : "..."
        },
        ...
    ],
    "children" : [
        {
            "_id" : "pm-tree",
            "name" : "Project Management (PM)",
            "description" : "...",
            "items" : [
                {
                    "image"       :       "./images/project-management-process-
diagram.png",
                    "name" : "Project Management Process Diagram"
                },
                {
                    "name" : "Objectives",
                    "table" : [
                        {
                            "name" : "Objective",
                            "description" : "Description"
                        },
                        {
                            "name" : "PM.O1",
                            "description" : "... "
```

```
                        }
                        ...
                    ]
                },
                {
                    "name" : "Activities",
                    "table" : [
                        {
                            "name" : "Activity",
                            "description" : "Description"
                        },
                        ...
                        {
                            "name" : "PM.4",
                            "description" : "Project Closure"
                        }
                    ]
                ...
            ],
            "children" : true
        },
        ...
        {
            "version" : "2014-02-11",
            "_id" : "products-tree",
            "name" : "Product description",
            "description" : "...",
            "items" : [
                {
                    "name" : "",
                    "table" : [
                        {
                            "name" : "Name",
                            "source" : "Source"
                        },
                        {
                            "name" : "Acceptance Record",
                            "source" : "Project Management"
                        },
                        ...
                        {
                            "name" : "Test Report",
                            "source" : "Software Implementation"
                        }
                    ]
                }
            ],
            "children" : true
        }
    ]
}
```

Lisäksi jos data olisi niin sanottua yksitasoista (flat data), jouduttaisiin ohjelmallisesti rakentamaan puun muotoinen hierarkia. Tämä puolestaan merkitsisi sitä, että JSON-

dokumentteihin jouduttaisiin lisäämään enemmän metadataa kuten tunnistetietoja lapsi- ja isäntäelementeistä.

Data on pilkottu useaan JSON-dokumenttiin, joista jokainen noudattaa saman tyyppistä rakennetta, kuin listauksessa 4.1 esitelty `vse-tree`-dokumentti.

MongoDB:en talletetut dokumentit

Ensimmäisenä MongoDB:iin talletetuista dokumenteista ladataan käyttöliittymän puuelementin muodon määrittelevä `vse-tree`, koska se on määritelty käyttöliittymän REST-toiminnallisuutta toteuttavassa komponentissa (JsonRest) haettavaksi puun juurielementiksi. Kuvassa 4.2 näkyvät puu ja sen elementit on luotu `vse-tree`-dokumentin (listaus 4.1) latauduttua käyttöliittymän komponenttiin.

MongoDB:n käynnistyminen

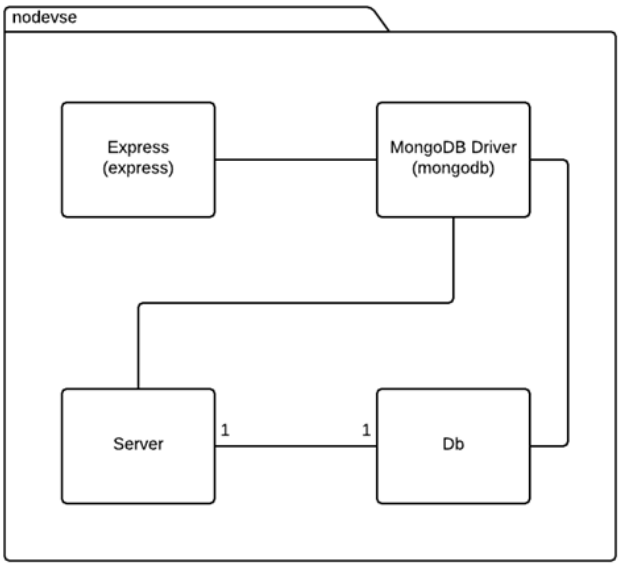
MongoDB:n taustaprosessi (`mongod.exe`) käynnistetään ensin, ellei sitä olla esimerkiksi asennettu Windows-alustaa käytettäessä Windows-palveluksi, jolloin se käynnistyy aina Windows-käyttöjärjestelmän käynnistyessä. Tällöin MongoDB on käytännössä käynnissä aina, ellei sitä erikseen sammuteta. MongoDB:n käynnistyminen ei lähetä Node tai web-sovelluksiin käynnistyessään mitään signaaleja, ainoastaan tietokannassa olevat dokumentit tulevat saataville.

4.3 Koneluettava rajapinta

Tässä kohdassa esitellään sovelluksen koneluettavan rajapinnan osuus, joka on toteutettu Node.js:n avulla. Toteutuksen mallina on käytetty Coenraetsin (2012) tekemää opasta REST-arkkitehtuurityylin mukaisen koneluettavan rajapinnan luomiseen käyttämällä Nodea, Expressiä sekä MongoDB:tä.

Node.js:n liittyvät tiedostot

Kaksi olennaisinta moduulia Node-palvelimen toteutuksessa ovat Express ja MongoDB-ajuri.



Kuva 4.8 - Node-sovelluksen moduulit

Kuvassa 4.8 on esitelty Node-sovelluksessa käytetyt moduulit. Express-web-ohjelmistokehystä käytetään tässä sovelluksessa REST-arkkitehtuurityylin mukaisen koneluettavan rajapinnan luomiseen. Rajapinta ottaa vastaan HTTP GET -kutsuja, jotka välitetään toiselle Node-moduulille MongoDB-ajurille, jonka avulla saadaan yhteys MongoDB-palvelimelle. Lopuksi rajapinta palauttaa pyydetyn JSON-dokumentin.

MongoDB-ajuri tarjoaa muiden muassa `Server` ja `Db` -moduulit. `Server`-moduulin avulla määritellään muiden muassa mistä osoitteesta ja portista MongoDB-palvelin on saavutettavissa. `Db`-moduulin avulla valitaan käytetty tietokanta, avataan yhteys tietokantaan sekä voidaan noutaa dokumentti. Listauksessa 4.3 on esitelty MongoDB-ajurin ohjelmakoodia tämän Node-sovelluksen osalta. Paketin nimi `nodevse` viittaa Node-projektikansioon, jossa ovat kaikki tämän Node-projektin tarvittavat tiedostot.

Express-moduulin käyttöönotto

Listaus 4.2 havainnollistaa Nodella ajettavan palvelimen toimintaa. Aluksi ladataan kaksi moduulia `require`-metodilla, Express web-ohjelmistokehys ja `guides.js`, jossa MongoDB-ajurin avulla otetaan yhteys tietokantaan (listaus 4.3). Jotta Express toimisi, se täytyy olla ladattu Noden työhakemistoon komennolla:

```
npm install express
```

Moduulien lataamisen jälkeen määritetään HTTP-kehyksiin lisää sääntöjä, jotka sallivat CORSin (Cross Origin Resource Sharing). Ongelmana tässä on se, että sovellus lähettää paikallisesti (`localhost`) ja HTTP-kehyksistä puuttuvat sallintakehykset, ja web-selaimet (Google Chrome ja Mozilla Firefox) eivät salli liikennettä ilman CORS-kenttiä.

Kentät saadaan kuitenkin lisättyä muutamalla koodirivillä. Jos tämän asetuksen jättää pois asiakassovellus ei saa pyytämiään JSON-dokumentteja rajapinnalta ja käyttöliittymän puuelementti ei lataudu. Konfiguraatiot saadaan asetettua Noden Express-sovellukseen `configure`-metodin sisällä `use`-metodilla.

Express tarjoaa sovellukselle `get`-metodin, jolla voidaan toteuttaa REST-arkkitehtuurityylin mukainen HTTP GET. Tässä tapauksessa esimerkiksi menemällä web-selaimella osoitteeseen `http://localhost:3000/guides/vse-tree` käynnistetään `findById`-metodi (listaus 4.2), joka hakee MongoDB:stä dokumentin, jonka `_id`-avaimen arvo on `vse-tree`. Selaimeen palautuu listauksessa 4.1 havainnollistettu `vse-tree`-dokumentin sisältö.

Listaus 4.2 - server.js toteutus

```
var express = require('express'),
    //ladataan guides.js
    epg = require('./routes/guides');

var app = express();

//cross origin resource sharing
var allowCrossDomain = function(req, res, next) {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE,OPTIONS');
    res.header('Access-Control-Allow-Headers',  'Content-Type,  Authorization,
Content-Length, X-Requested-With');

    // kaapataan OPTIONS-metodi
    if ('OPTIONS' == req.method) {
      res.send(200);
    }
    else {
      next();
    }
};

app.configure(function () {
    app.use(allowCrossDomain);
    app.use(express.logger('dev'));
    app.use(express.bodyParser());
});

...
app.get('/guides/:id', epg.findById);

app.listen(3000);
console.log('Listening on port 3000...');
```

Lopuksi ohjelmakoodissa on määritelty Node-sovellus kuuntelemaan porttia 3000 Expressin tarjoamalla `listen`-metodilla.

MongoDB-ajurin käyttöönotto

Yhteys MongoDB-dokumenttitietokantaan saadaan Noden MongoDB-ajurilla (MongoDB, Inc. 2014f). Ajuri saadaan käyttöön `require`-kutsulla, mutta mongodb-niminen Node-moduuli on täytynyt ensin ladata NPM:sta komennolla:

```
npm install mongodb
```

Server-moduulin uusi ilmentymän parametreiksi määritellään MongoDB-palvelimen olevan paikallisessa konessa (`localhost`) portissa 27017. Seuraavaksi määritellään käytettäväksi tietokannaksi `test` ja palvelimeksi juuri määritelty `server`.

Seuraavaksi avataan yhteys `test`-tietokantaan `open`-metodilla. Tämä jälkeen otetaan yhteys `vsetest`-kokoelmaan `collection`-metodilla. Node antaa konsoliin virheviestin, jos haluttua kokoelmaa ei löydy.

Varsinaisen toiminnallisuuden dokumentin noutamisesta toteuttaa `findById`-metodi, joka saa parametritietona haettavan dokumentin `_id`-arvon. Seuraavaksi täsmennetään noudettavan dokumentin olevan `vsetest`-kokoelmassa, ja dokumenttia pyydetään tietokannasta MongoDB:n omalla `findOne`-metodilla.

Listaus 4.3 - guides.js toteutus

```
var mongo = require('mongodb');

var Server = mongo.Server,
Db = mongo.Db;
//BSON = mongo.BSONPure;

var server = new Server('localhost', 27017, {auto_reconnect: true});
db = new Db('test', server);

db.open(function(err, db) {
    if(!err) {
        console.log("Connected to 'test' database");
        db.collection('vsetest', {strict:true}, function(err, collection) {
            if (err) {
                console.log("The 'vsetest' collection doesn't exist...");
            }
        });
    }
});

exports.findById = function(req, res) {
    var id = req.params.id;
    console.log('Retrieving guide: ' + id);
    db.collection('vsetest', function(err, collection) {
        //collection.findOne({'_id':new    BSON.ObjectID(id)},    function(err,
item) {
```

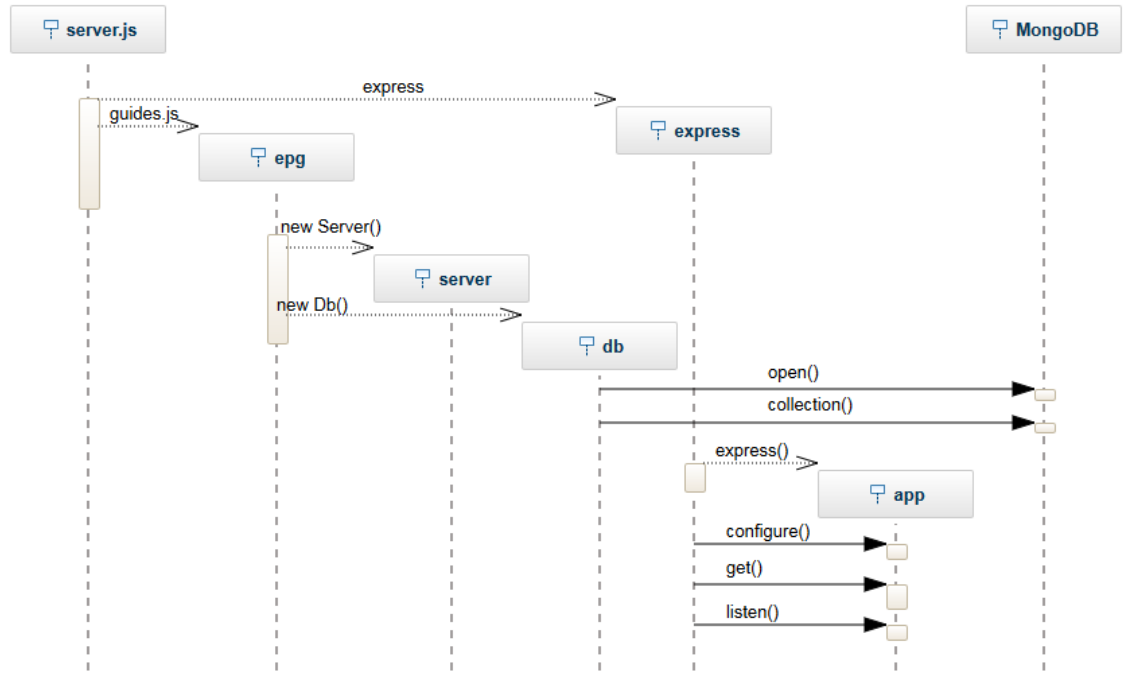
```
        collection.findOne({'_id': id}, function(err, item) {
            res.send(item);
        });
    });
};

...
```

Listauksessa 4.3 oleva MongoDB-ajurin moduuli `BSON` on kommentoitu esimerkkisyistä ohjelmakoodiin. Tämän moduulin avulla on mahdollista käsitellä MongoDB:n oletuksena dokumenteissa käyttämiä ObjectID-tyyppisiä arvoja. Tässä sovelluksessa on kuitenkin itse määritellyt `_id`-avaimien arvot ja `BSON`-moduulia ei ole tarvinnut ottaa käyttöön. Ohjelmakoodissa on myös kommentoituna esimerkki, kuinka haetaan ObjectID:tä käyttävä dokumentti MongoDB:sta.

Node-palvelimen käynnistyminen

Node-sovellus ottaa käynnistymisensä aikana yhteyden MongoDB-dokumenttitietokantaan, joten sen on oltava saavutettavissa Node-sovelluksen käynnistyessä. Node-sovelluksen toimintaa on jo jonkin verran esitelty otettaessa Express-web-ohjelmistokehystä sekä Noden MongoDB-ajuria käyttöön (listaukset 4.2 ja 4.3). Kuvassa 4.9 on esitettynä Node-palvelinalustalla ajettavan sovelluksen käynnistyminen sekvenssikaaviona.



Kuva 4.9 - Sekvenssikaavio Node-palvelimen sovelluksen käynnistymisestä

Node-sovelluksen käynnistyessä ladataan `express`-Node-moduuli `express`-muuttujaan ja `guides.js`-tiedosto `epg` muuttujaan, samalla suoritetaan `guides.js`-tiedoston ohjelmakoodi. Suoritus jatkuu luomalla `Server`- ja `Db`-moduuleille

uudet ilmentymät `server` ja `db`. Tämän jälkeen otetaan yhteys MongoDB:n tietokantaan `db`:n `open`-metodilla ja valitaan kokoelma `db`:n `collection`-metodilla. Kun yhteys on saatu, jatkuu koodin suoritus luomalla `app`-express-sovellus `express`-metodilla. Lopuksi määritellään `express`in tarjoamat metodit `configure`, `get` ja `listen`, `app` -express-sovellukseen.

4.4 Dataan mukautuva käyttöliittymä

Tässä kohdassa esitellään sovelluksen mukautuva käyttöliittymä, joka on toteutettu JavaScript-pohjaisella Dojo-työkalupakilla. Toteutettu käyttöliittymä mukautuu koneluettavalta rajapinnalta pyydettyjen JSON-dokumenttien rakenteeseen. Mukautuva käyttöliittymä on toteutettu käyttämällä useita eri Dojo-oppaita.

Web-sovellukseen liittyvät tiedostot ja sen käyttöönotto

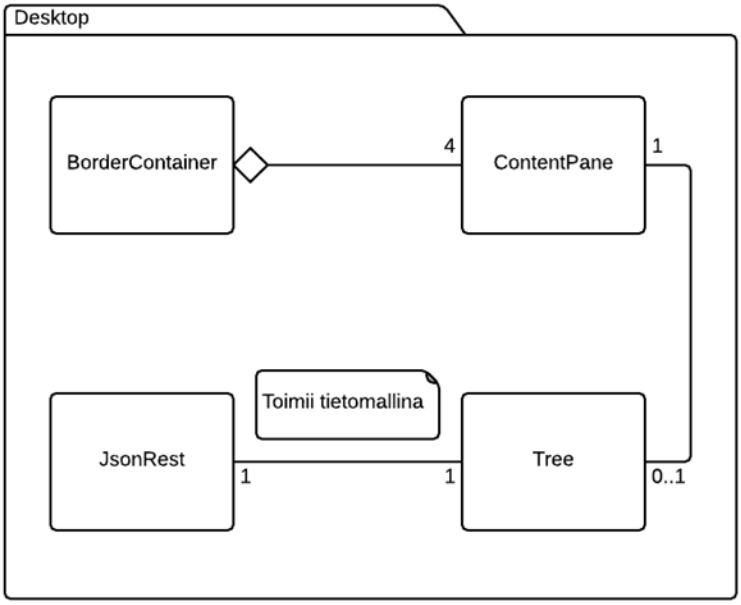
Listauksessa 4.4 on esitelty web-sovelluksen HTML-merkkaus (html markup) index.html-tiedoston osalta. Web-sovelluksen käynnistyessä kutsutaan käyttöliittymän käynnistymiseen vaadittavia moduuleja. Näistä tärkein on `Desktop`, joka itse rakennettu pienoisohjelma. Tämän pienoisohjelman sisälle on rakennettu elektronisen prosessioppaan toiminnallisuus ja ulkoasu.

Listaus 4.4 - index.html:n html-merkinnät

```
<!DOCTYPE html>
<html >
    <head>
        <meta charset="UTF-8">
        <title>EPG</title>
        ...
        <link rel="stylesheet" href="./dijit/themes/soria/soria.css">
        <style type="text/css">
            ...
        </style>
        <script src="./dojo/dojo.js"></script>
        <script>
            require([
                "dojo/ready", "js/Desktop", "dojo/_base/window"
            ], function(ready, Desktop, win){
                ready(function(){
                    var desktop = new Desktop();
                    desktop.placeAt(win.body()).startup();
                });
            });
        </script>
    </head>
    <body class="soria">
    </body>
</html>
```

Suoritus alkaa asettamalla Dojo-moduulit muuttujiin, joista esimerkiksi `dojo/ready` rekisteröi funktion ajoon, kun DOM on valmiina ja kaikki ratkaisemattomat `require`-kutsut, sekä muut korkeammilla prioriteeteilla olevat funktiot, on ratkaistu. (Dojo Foundation 2014k)

Kun tarvittavat Dojo-moduulit on ladattu, luodaan `Desktop`-muuttujaan `Desktop`-moduulin uusi ilmentymä. Tämän jälkeen käynnistetään `Desktop`-pienoisohjelma kutsumalla sen `startup`-metodia. Lopuksi pienoisohjelma asetetaan selain-ikkunan `body`-osioon käyttämällä `window`-moduulin ominaisuuksia. (Dojo Foundation 2014b, 2014h)



Kuva 4.10 - Web-sovelluksen moduulit

Kuvassa 4.10 on esitettynä web-sovelluksen moduulit, jotka on määritelty `Desktop`-pieoisohjelman toteutuksessa. Web-sovelluksessa käytettyjä näkyviä Dijit-komponentteja ovat `BorderContainer`, `ContentPane` ja `Tree`. Käyttäjälle näkymättömiä komponentteja on yksi, `JsonRest`, joka hoitaa REST-arkkitehtuurityylin mukaisen yhteyden koneluettavaan rajapintaan. Web-sovelluksen käyttöliittymässä on käytetty `BorderContainer`-komponenttia, joka koostuu neljästä `ContentPane`-komponentista. Yhden `ContentPane`-komponentin sisälle on sijoitettu puu-elementti `Tree`. `JsonRest` toimii `Tree`-komponentin tietomallina.

Desktop-pienoisohjelma

Tässä määritellään `Desktop`-moduuli, joka on Dojon pienoisohjelma (widget). Mallina toteutukseen on käytetty Dojon dokumentaatiosta löytyviä oppaita (Dojo Foundation 2014f, 2014n).

Listauksessa 4.5 on esitetty oman `Desktop`-pienoisohjelman toteutusta. Aluksi määritellään `define`-kutsulla riippuvuudet, johon listataan kaikki käyttöliittymän toiminnalle välttämättömät moduulit ja siirretään niiden sisältö muuttujiin. Esimerkiksi `dijit/layout/BorderContainer` sisältö siirtyy muuttujaan `BorderContainer` ja niin edelleen. Käyttöliittymämalliksi (template) on asetettu `desktop.html`-tiedoston sisältö `templateString`-attribuutilla.

Listaus 4.5 - Desktop-moduulin (Desktop.js) kooditoteutus

```
define([
    "dojo",
    "dojo/_base/declare",
    ...
    "dojo/store/JsonRest",
    "dijit/Tree",
    "dijit/registry",
    "dijit/layout/BorderContainer",
    "dijit/layout/ContentPane",
    "dojo/text!./templates/desktop.html"
], function(
dojo,
declare,
...
JsonRest,
Tree,
registry,
BorderContainer,
ContentPane,
template){

    return declare("Desktop", [BorderContainer, ContentPane, ...], {
        templateString: template,
        widgetsInTemplate: true,
        ...
        // Kun desktop-malli on ladattu. (Huom. Lapsielementit ei
        // välttämättä valmiina!)
        postCreate: function(){
            ...
        },
        //Kun desktop-mallin lapsielementit on ladattu (mm. BorderContainer)
        startup: function() {
            ...

            var reststore = new JsonRest({...});

            var myTree = new Tree({...}, this.containerNode);
            //viittaa data-dojo-attach-pointiin desktop.html -mallissa
        },
        resize: function() {
            ...
            registry.byId("borderContainer").resize();
        }
    });
});
```

Uudet ilmentymät luodaan `startup`-metodin alla, koska Dojon pienoisohjelmadokumentaatiossa suositellaan käyttämään `startup`-metodia käyttöliittymäkomponenttien kanssa. Metodi varmistaa, että esimerkiksi käyttöliittymä komponentit ovat ladattu saataville. Tämä on tärkeää, koska esimerkiksi käyttöliittymän mallissa viitataan `containerNode`-avainsanan avulla. (Dojo Foundation 2014i)

Ikkunankoko piirretään uudelleen aina kun kutsutaan `resize`-metodia. Tässä on käytetty hyväksi Dijitin `registry`-moduulia, joka pitää ajonaikaista kirjaa kaikista käyttöliittymän näkyvistä komponenteista. (Dojo Foundation 2014k)

Pienoisohjelman malli

Listauksessa 4.6 on esitelty `Desktop`-pienoisohjelman mallin HTML-merkinnät (desktop.html). Mallissa ei ole HTML-sivuilla normaalisti olevia otsaketietoja, vaan `desktop.html` -tiedoston sisältö asetetaan lopulta `index.html` -sivun body-osioon. Toteutuksessa on mukailtu Dojon oppaista löytyviä BorderContainer-komponentin käyttöön liittyviä esimerkkejä (Dojo Foundation 2014j).

Listauksessa 4.6 HTML-merkinnät on niin sanotusti paketoitu yhdellä ylimääräisellä div-osiolla. Tämä tehtiin sen takia, että ohjelmoidessa huomattiin Dojon karsivan tämän tiedoston käyttöönotossa ylimmän tunniste-osion pois. Eli tässä tapauksessa Dojon mekanismit karsivat pois `<div class=”topClass”>…</div>` -parin pois.

Seuraavana on vuorossa `BorderContainer`, joka on määritelty käyttöön `data-dojo-type` -attribuutilla. `BorderContainer`-komponenttiin on määritelty myös `id` -kenttä, johon päästään käsiksi esimerkiksi `registry`-moduulia käyttämällä. Lisäksi mukana on `data-dojo-attach-point` -attribuutti, joka mahdollistaa tunnisteeseen käsiksi pääsyn JavaScript-koodista käsin. (Dojo Foundation 2014f)

BorderContainer koostuu ContentPane-komponenteista, jotka merkitään `data-dojo-type` -attribuutilla. Niissä on myös `data-dojo-props` -attribuutti, jossa voidaan ilmaista kyseessä olevan `ContentPane`n sijainti `BorderContainer`-komponentissa. Kuvassa 4.2 nähdään, että käyttöliittymä koostuu neljästä eri osasta: ylä- ja alapaneelista, sekä kahdesta paneelista sivun keskellä.

Yläpaneelin (`top`) sisältönä ovat samalla rivillä kuva vasemmalla ja tekstiä oikealla. Toteutuksessa kummallekin on luotu oma div-osio ja tasaukset oikealle ja vasemmalle on toteutettu css-tyyleillä. Alapaneelin (`bottom`) toteutus on samantyyppinen kuin yläpaneelin ainoana poikkeuksena on, että vasemmalla on tekstiä ja oikealla on tekstiä sekä kuva.

Vasemman paneelin (`leading`) määrittelyssä on otettu käyttöön `splitter`-paneelienjakaja, jolla käyttäjä voi itse säätää web-sovelluksen vasemman paneelin leveyttä. Rakenteellisesti vasemman paneelin sisälle on asetettu div-osio, johon asetetaan Tree-elementti. Asettaminen on havainnollistettu listauksessa 4.5, jossa viitataan `containerNode`-viittaustietoon.

Listaus 4.6 - desktop.html -mallin (template) koodi

```
<div class="topClass">
    <div id="borderContainer"
            data-dojo-type="dijit/layout/BorderContainer"
            data-dojo-attach-point="bcPane" style="..." >
        <div data-dojo-type="dijit/layout/ContentPane"
            data-dojo-props="region:'top'"
            data-dojo-attach-point="topPane">
            ...
        </div>
        <div data-dojo-type="dijit/layout/ContentPane"
            data-dojo-props="splitter:true, region:'leading'"
            data-dojo-attach-point="leadingPane">
            <div data-dojo-attach-point="containerNode"></div>
        </div>
        <div id="contentId"
            data-dojo-type="dijit/layout/ContentPane"
            data-dojo-props="region:'center'"
            data-dojo-attach-point="centerPane">
        </div>
        <div id="bottomContainerId"
            data-dojo-type="dijit/layout/ContentPane"
            data-dojo-props="region:'bottom'"
            data-dojo-attach-point="bottomPane">
            ...
        </div>
    </div>
</div>
```

Keskipaneelin (`center`) määrittelyissä on id-attribuutti, johon viitataan käyttämällä `dojo`n id-hakutoimintoa, kun siirretään tietoa keskipaneeliin käyttäjän näkyville.

REST-toiminnallisuus

JsonRest on HTTP-pohjaisen asiakasohjelman, joka on täyden RESTin (RESTful), kevyt objektimuistitoteutus. Listauksessa 4.7 on esitelty JsonRest-komponentin uuden ilmentymän `reststoren` ominaisuudet ja metodit `target`, `mayHaveChildren`, `getChildren`, `getRoot` ja `getLabel`. (Dojo Foundation 2014d, 2014m)

Datan rajapinnan osoite ilmoitetaan `target`-ominaisuuden avulla. Koska saatava JSON-data on puun muotoista, sen käsittelyyn tarvitaan metodeja. Metodeilla käydään läpi haetun elementin mahdollisia lapsielementtejä (`mayHaveChildren` ja `get-`

Children). Datamalliin on määritelty sen juurielementti getRoot-metodilla. Myöhemmin luotavan puuelementin näkyvään osaan haetaan otsikkotiedot getLabel-metodilla.

Listaus 4.7 - reststoren toiminnot

```
var reststore = new JsonRest({
    target: "http://localhost:3000/guides/",
    mayHaveChildren: function(object){
       // katsotaan, löytyykö children
       return "children" in object;
    },
    getChildren: function(object, onComplete, onError){
        // this.get kutsuu 'mayHaveChildren', jos palauttaa true, ladataan
        //tarvittavat, jolloin true ylikirjoitetaan { item } -muotoon
        this.get(object._id).then(function(fullObject){
            // kopioidaan alkuperäiseen objektiin, jolloin se sisältää
            //myös lapsielementit (children)
            object.children = fullObject.children;
            // kun on saatu täysi objekti, pitäisi olla saatavilla
            //taulukollinen lapsielementtejä (children)
            onComplete(fullObject.children);
         }, function(error){
            // virhe tapahtui, se kirjattiin ja merkittiin ettei
            //lapsielementtejä ole saatavilla
            console.error(error);
            onComplete([]);
         });
    },
    getRoot: function(onItem, onError){
         // haetaan juuriobjekti, tehdään get() ja vaste callbackataan
         this.get("vse-tree").then(onItem, onError);
         },
    getLabel: function(object){
         // haetaan nimi (jotkut mallit käyttävät 'labelAttr', joka on
         //avaimena datassa, tässä haetaan vain avaimen 'name' arvo)
         return object.name;
    }
});
```

Lisäksi käyttöliittymään tuleva Tree-komponentti vaati, että tietomallissa on mayHaveChildren, getChildren, getRoot ja getLabel -metodit, jotka hahmottavat datan puuksi. Listaus 4.7 on mukaelma Dojo oppaasta (2014e), jossa on esimerkkejä JsonRest-komponentin käytöstä.

Käyttöliittymän puuelementti

Listauksessa 4.8 on esitelty käyttöliittymän Tree-komponentin ilmentymän ominaisuudet ja siihen liittyvät tapahtumakäsittelijät (model, showRoot, onLoad ja onClick). Nämä on määritelty Dojon API dokumentaatiossa (2014c) ja listauksessa 4.8 on esitetty niiden toiminnallisuus. Ominaisuuksiin merkitään mitä tietomallia käytetään,

ja tähän on laitettu aiemmin luotu JsonRest-komponentin ilmentymä (`reststore`). Ominaisuudella `showRoot` on ilmoitettu, että pidetään puun juurisolmu näkyvissä.

Kun puukomponentti on lakannut lataamasta dataa Node-rajapinnasta, BorderContainer-komponentin ikkunakoot säädetään uudelleen. Viittaus BorderContainer-komponenttiin haetaan `registry`n avulla Dijitin ajonaikaisesta rekisteristä, jotta paneelien uudelleenpiirtokäsky voidaan toimittaa. Tapahtumankäsittelijä `onLoad`ia kutsutaan, kun puu on lakannut lataamasta.

Listaus 4.8 - myTreen toiminnot

```
var myTree = new Tree({

    model: reststore, //JsonRest
    showRoot: true,

    //onLoad() kutsutaan kun puu lopettaa lataamisen ja laajenemisen.
    onLoad: function() {


        //haetaan widgettiä dijit-rekisteristä id-kentän mukaan
        registry.byId("borderContainer").resize();
        },

    //onClick(item, node, evt) Callback, kun puun elementtiä klikataan
    onClick: function(item) {...}

}, this.containerNode); //viitataan data-dojo-attach-pointiin desktop.html
sivulla
```

Toteutettuna on myös puun solmujen klikkauksia seuraava tapahtumankäsittelijä `onClick`-metodi, jonka toteutusta on esitelty listauksessa 4.9.

Tree-komponentin tapahtumakäsittelijä

Listauksessa 4.9 on esitelty `Tree`-komponentin onClick-tapahtumakäsittelijään liitetty toiminnallisuus. Klikattavasta elementistä haetaan avainsanoja ja toiminto määräytyy avainsanan mukaan. Tarkoituksena on luoda sisältö käyttöliittymän keskimmäiseen (center) paneeliin puuelementtipaneelin (puuelementti sijaitsee leading-paneelissa) viereen.

Listaus 4.9 - myTreen onClick-kuuntelijan toiminnallisuus

```
//onClick(item, node, evt) Callback, kun puun elementtiä klikataan
onClick: function(item) {

    var str = item.description;

    //katsotaan onko "undefined"
```

```
    if(!(typeof str==="undefined")){
        var res = str.replace(new RegExp('\r?\n','g'), '<br />');
    }

    var content = "<h1>" + item.name + "</h1><p>" + res + "</p>";

    for(var i=0; i< item.items.length; i++){
        //löytyykö 'name' -key
        if (!(typeof (item.items[i].name)==="undefined")){
            content += "<h3>" + item.items[i].name + "</h3>";
        }
        ...
        //löytyykö 'image' -key
        if (!(typeof (item.items[i].image)==="undefined")){
            content += "<img src='" + item.items[i].image + "'/>";
        }
        //löytyykö 'table' -key
        if (!(typeof (item.items[i].table)==="undefined")){
            content += "<table style='width:100%'>";
            for(var j=0; j< item.items[i].table.length; j++){
                if (j % 2 === 0 ){
                    content += "<tr class='alt-row'>";
                } else {
                    content += "<tr>";
                }
                //löytyykö tablesta 'name' -key
                if (!(typeof (item.items[i].table[j].name)==="undefined")){
                    if(j===0) {
                       content += "<th>" + item.items[i].table[j].name
                       + "</th>";
                    } else {
                       content += "<td>" + item.items[i].table[j].name
                       + "</td>";
                    }
                }
                //löytyykö tablesta 'description' -key
                if                                                    (!(typeof
(item.items[i].table[j].description)==="undefined")){
                    if(j===0) {
                       content += "<th>" + item.items[i].table[j].description
                       + "</th>";
                    } else {
                       var s = item.items[i].table[j].description;
                       var r = s.replace(new RegExp('\r?\n','g'), '<br />');
                       content += "<td>" + r + "</td>";
                    }
                }
                ...
                content += "</tr>";
            }
            content += "</table>";
        }
    }
    dojo.byId('contentId').innerHTML = content;
}
```

Ensin haetaan klikatun elementin juuri tasolta avainsanoja, joiden arvot asetetaan sisällön otsikoksi ja tämän alle tulevaksi johdannoksi. Sivun sisältö ketjutetaan `content`-muuttujaan, jonka sisältö asetetaan lopuksi käyttöliittymän keskimmäiseen paneeliin. Aina ensin tarkastetaan onko haluttua avainta elementissä käyttämällä hyväksi JavaScriptin `typeof`-operaattoria.

JSON-dokumentissa rivivaihto on merkitty tekstin sekaan `\n` -merkinnöillä. Esimerkiksi kohdalle tulee `description`-avain, sen arvo käydään läpi RegExpillä ja vaihdetaan kaikki löytyvät `\n` -merkinnät HTML:n `<br />` -rivinvaihtotunnisteilla.

Kaiken muun sivulle tulevan sisällön on tarkoitus löytyä JSON-dokumentin `items`-taulukosta. Klikatun elementin `items`-taulukossa olevat avaimet voivat olla mm. `name`, `image` tai `table`. Tällä syvemmällä tasolla `name`-avaimen arvosta tulee alemman tason `<h3>` -otsikko, `image`-avaimen arvona on halutun kuvatiedoston polku, ja puolestaan `table`-avaimen kohdalla käyttöliittymään piirretään taulukko sisällytetyistä arvoista. Taulukko piirretään käyttöliittymään niin, että se on koko vapaan tilan levyinen ja joka toisella rivillä on harmaa taustaväri. Taulukon muotoisessa arvossa voi olla samoja avaimia, kuin löytyi ylemmiltä tasoilta. Ja avaimiin liittyvät toiminnot ovat samoja kuin ylemmillä tasoilla ja arvot ketjutetaan jälleen `content`-muuttujan sisään. Lopuksi asetetaan rivin sulku- ja taulukon sulkutunnisteet.

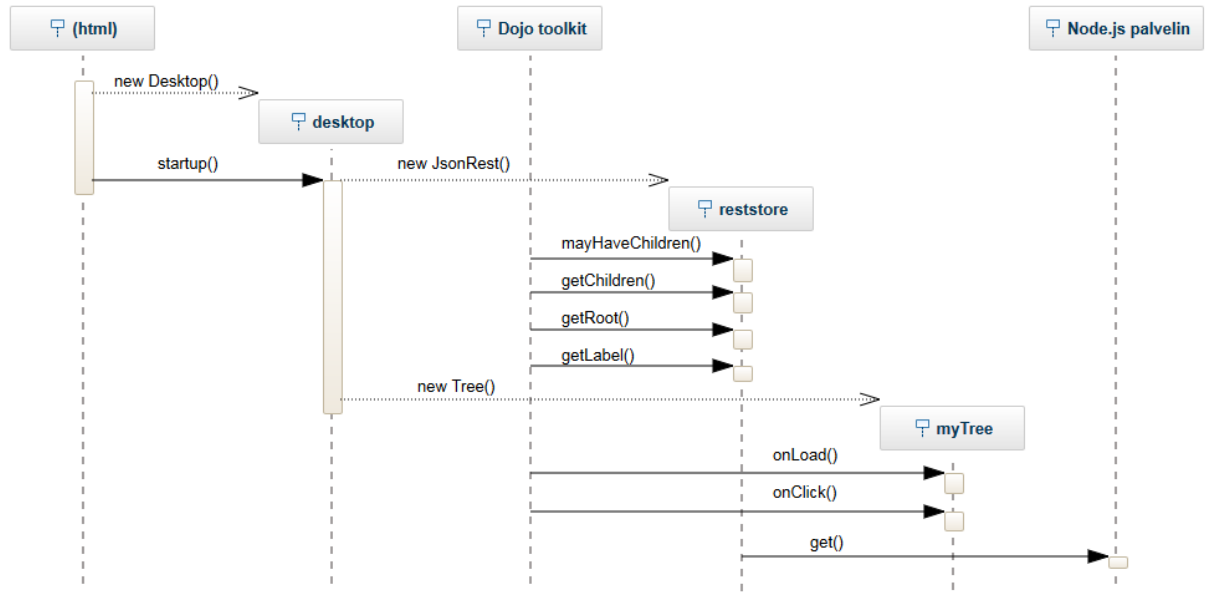
Aivan `onClick`-tapahtuman lopuksi asetetaan `content`-muuttujan arvo selain-ikkunan keskipaneeliin, joka haetaan `contentId` -tunnistetiedon perusteella. Tunnistetieto viittaa Desktop-pienoisohjelman mallissa (`desktop.html`) olevaan `id`-tietoon. Elementin sisältöä on esitelty listauksessa 4.1, jossa havainnollistetaan elementin ja JSON-dokumentin rakennetta.

Dojo web-sovelluksen käynnistyminen

Dojo web-sovellus ottaa yhteyden Node-palvelimen sovellukseen, joka puolestaan ottaa käynnistymisen aikana yhteyden MongoDB:en. MongoDB:n ja Node-sovelluksen tulee olla saavutettavissa web-sovelluksen käynnistyessä.

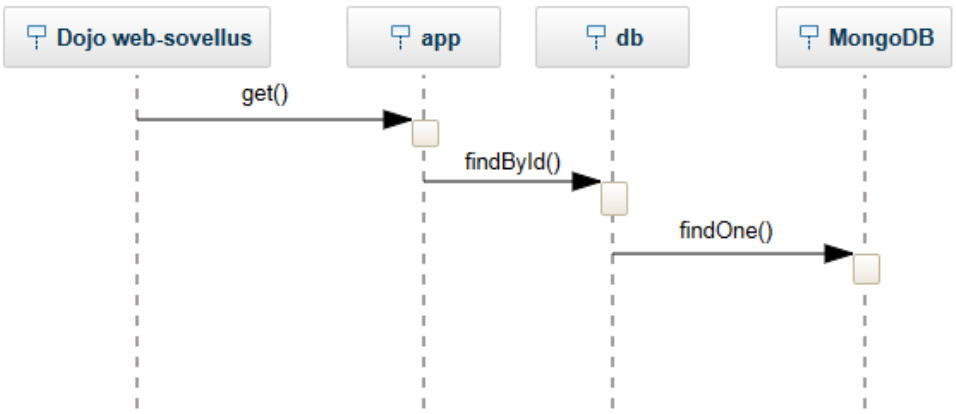
Kuvissa 4.11 ja 4.12 on esitetty sekvenssikaavioina Dojo web-sovelluksen käynnistyminen. Kun selain-ikkuna aukeaa, muodostetaan itse määritellystä `Desktop`-moduulista (oma pienoisohjelma) uusi ilmentymä `Desktop`. Tämän jälkeen kutsutaan `Desktop`in `startup`-metodia. Metodissa luodaan `JsonRest` -Dojo-moduulista uusi ilmentymä `reststore`, jonne toteutetaan Dojon tarjoamat metodit `mayHaveChildren`, `getChildren`, `getRoot` ja `getLabel`. Seuraavaksi suoritus jatkuu luomalla `Tree`-komponentin uusi ilmentymä `myTree`, jonka tietomalliksi liitetään `rests-`

`tore`. `Tree`-komponentin uuteen ilmentymään toteutetaan Dojon tarjoamat metodit `onLoad` ja `onClick`.



Kuva 4.11 - Sekvenssikaavio web-sovelluksen käynnistymisestä

Kun web-sovelluksen käyttöliittymäkomponentit ovat latautuneet selain-ikkunassa olevaan käyttöliittymään, `reststore` lähettää `get`-kutsun Node-sovellukselle, jonka avulla se pyytää `vse-tree`-dokumenttia käyttöliittymän puuelementin juureksi. Dojo web-sovellukselta saapuvan `get`-kutsun vastaanottaa Express-sovellus `app`. Tästä `vse-tree`-dokumentin pyytäminen siirtyy MongoDB-ajuri `db`:lle `findById`-kutsulla, joka muuntaa sen MongoDB:n ymmärtämään `findOne`-kutsuun.



Kuva 4.12 - Sekvenssikaavio Node-palvelimen tapahtumista, kun web-sovellus käynnistetään

Kun Dojo web-sovellus on saanut käynnistyksessä pyytämänsä dokumentin, `rests-tore` tulkitsee sen `Tree`-komponentin ilmentymän ymmärtämään muotoon, ja puu latautuu selain-ikkunan vasempaan (leading) paneeliin.

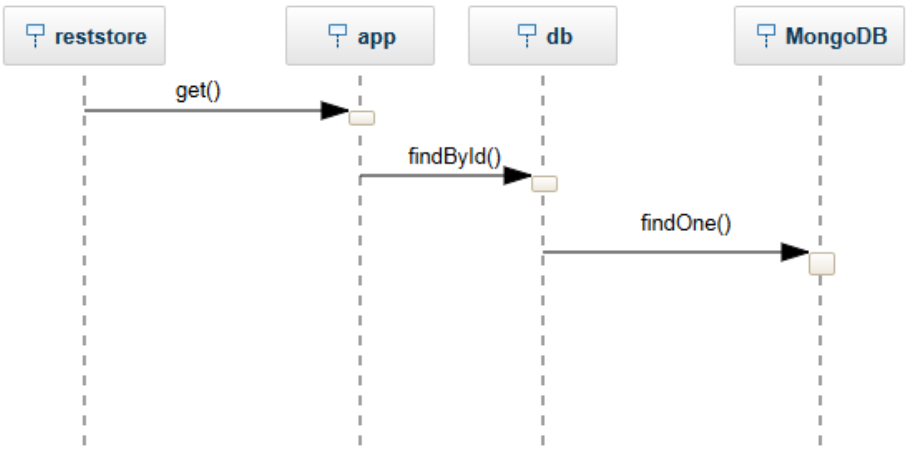
Käyttäjä klikkaa käyttöliittymän elementtiä

Listauksessa 4.7 JsonRest -komponentin toteutuksessa elementtien nimiksi puuhun haettiin name-avaimien arvot. Käyttäjä klikkaa auki Product description -solmun, joka on `vse-treen` alidokumenttina `children`-taulukossa (listaus 4.1). Product description -elementin `_id`-avain on `products-tree`, elementistä löytyy muiden muassa `name`, `items` ja `children` -avaimet.

Avaimen `items` sisältö siirretään selain-ikkunan keskipaneeliin, kun puun Product description -elementtiä klikataan. `Children`-avaimella kerrotaan, että Product descriptions -elementtiin liittyvää dataa on lisää tarjolla. Lisää dataa pyydetään puolestaan `products-tree` -avaimella ja MongoDB-palvelimelta tulisi löytyä `products-tree` -dokumentti (dokumentin juuressa on `"_id"="products-tree"` avain/arvo-pari).

Käyttäjä avaa puusta Product descriptions -elementin solmun ja `reststore` lähettää `get`-kutsun Node.js -palvelimelle `products-tree` -dokumentista REST-arkkitehtuurityylin mukaisesti koneluettavalle rajapinnalle osoitteeseen `http://localhost:3000/guides/products-tree`.

Node-sovellus kuuntelee porttia 3000 HTTP GET -metodin varalta ja sellaisen saapuessa, pyyntö ohjataan `db`:lle `app`:n `get`-kutsulla ja `products-tree` merkitään tunnistetiedoksi. `db`:n `findById`-metodi pyytää `products-tree` `_id`-avaimella olevaa dokumenttia MongoDB:lta sen omalla `findOne`-kutsulla (kuva 4.13).



Kuva 4.13 - Sekvenssikaavio käyttäjän klikatessa puun solmu auki

Kun pyydetty `products-tree`-dokumentti on saatu `reststore`en, käyttöliittymän puuelementti päivitetään uudella sisällöllä ja Product description-elementin lapsielementit tulevat näkyviin.

5 YHTEENVETO JA ARVIOINTI

Tutkimuksen tavoitteena oli luoda elektroninen prosessiopas käyttämällä ja esittelemällä vapaasti saatavilla olevia työkaluja sekä teknologioita. Tarkoitus oli selvittää miten avoimia tietovarantoja hyödynnetään teknisesti.

Tutkimus aloitettiin perehtymällä pienten ohjelmistoyksiköiden prosessistandardin Entry-tason oppaaseen ISO/IEC TR 29110-5-1-1:2012(E), jonka oli tarkoitus toimia kaksijakoisessa roolissa elektronisen prosessioppaan testidatana sekä prosessioppaana ohjelmistototeutuksen osalta.

Prosessistandardi yritettiin muuttaa elektronisen prosessioppaan testidataksi mallintamalla sitä relaatiomalliin, koska sovelluksen tiedonhallintajärjestelmänä oli aluksi tarkoitus käyttää relaatiotietokantaa. Kuitenkin huomattiin, että prosessistandardin saattaminen relaatiomalliin vaati huomattavaa vaivannäköä. Liitteessä 5 on esitelty kesken jäänyt ER-kaavio relaatiomalliin saattamisesta.

Relaatiomallin tilalle vaihdettiin dokumenttimalli, jossa voi olla sisäkkäisiä rakenteita ja prosessistandardin rakenne tuntui sopivan tähän malliin. Esimerkiksi prosessin sisällä on aktiviteetteja, joten dokumenttimallissa voidaan aktiviteetti-dokumentit sijoittaa prosessi-dokumentin sisälle. Dokumenttitietokannaksi valittiin MongoDB, joka tallettaa tiedon BSON-muotoisena. Koska BSON on periaatteessa JSON-tiedonsiirtoformaatti muutamalla lisäyksellä, dokumentit voidaan tallettaa MongoDB:en täysin JSON-muotoisena.

JSON-formaatin käyttö MongoDB:n kanssa sopikin hyvin tehdyn työn kannalta, sillä sitä voidaan myös käyttää tiedonsiirtoformaattina rajapintojen välillä. Koneluettavaa rajapintaa ei kuitenkaan lähdetty heti tämän jälkeen toteuttamaan vaan työpanos fokusoitiin mukautuvaan käyttöliittymään, joka on Dojo toolkitilla toteutettu web-sovellus. Alkuvaiheessa web-sovellus luki prosessistandardista luotuja JSON-tiedostoja suoraan tiedostosta. Näin voitiin aloittaa asiakasohjelman toteutus ilman, että testidataa tarvitsisi hakea rajapinnan kautta tietokannasta. Samalla huomattiin, että prosessistandardista luotuihin JSON-tiedostoihin tarvitsi lisätä metatietoa, jota käyttöliittymän komponentit käyttivät hyväkseen käyttöliittymää luodessaan.

Koneluettava rajapinta toteutettiin noudattamalla REST:n neljää perusperiaatetta. Nämä periaatteet ovat: HTTP-metodeja tulisi käyttää eksplisiittisesti, palvelun tulisi olla tila-

ton, URI:t tulisi näyttää hakemistokaltaisina ja tiedonsiirtoformaattina käytettäisiin XML:a, JSON:ia tai molempia. Näitä perusperiaatteita oli tarkoitus seurata, kun toteutettiin koneluettava rajapinta käyttämällä Node.js-alustaa, sekä siihen liitettäviä kolmannen osapuolen moduuleja Expressiä ja MongoDB-ajuria.

Rajapintaan onnistuttiin saamaan RESTin toiminnallisuus muutamalla rajoituksella. Rajapinnan avulla voidaan ainoastaan hakea dokumentteja tietokannasta, eli HTTP-metodeista käytössä on ainoastaan GET. Sovellus ja rajapinta eivät myöskään pidä yllä mitään tilan kirjausta, eli kun dokumentti pyydetään, pyydetään tietyllä tunnistetiedolla oleva dokumentti. URI:sta on myös tehty hakemistokaltaisia esimerkiksi, jos kirjoitetaan selaimeen http://localhost:3000/guides/vse-tree antaa rajapinta tietokannasta vse-tree-tunnistetiedolla varustetun dokumentin tietokannasta. Dokumentit ovat JSON-muotoisia, joten tiedonsiirtoformaattikriteeri on myös täytetty.

Rajapinnan määrittelyn jälkeen asiakasohjelman toteutuksessa huomattiin puutteita ja sovellukseen jouduttiin tekemään muutoksia. Sovelluksen ensimmäiseen versioon valituilla komponenteilla ilmeni ongelmia mm. ikkunanpiirtämisessä. Uusi versio web-sovelluksesta toteutettiin enemmän Dojon hengessä, siitä tehtiin räätälöity pienoisohjelma ja sovelluksen komponentit saatiin toimimaan keskenään paremmin.

Kehitystarpeet

Elektronisen prosessioppaan sovelluksen toteutuksessa olisi vielä parannettavaa ja muutamia ominaisuuksia jäi uupumaan. Prosessioppaassa näkyvät kuvat eivät tällä hetkellä sijaitse MongoDB-tietokannassa vaan niiden resurssit löytyvät asiakasohjelman yhteydestä ja tietokannasta saatavissa dokumenteissa on viittaus kuvaresurssin osoitteeseen. Tämän voisi pitää ennallaan, jos kuvat haettaisiin vaikka jostakin WWW-osoitteesta, tai vaihtoehtoisesti kuvat voisi tallettaa dokumentteihin tekstimuotoisena Base64-koodattuna (liitteessä 8 on esitelty kuva ja sen Base64 koodi). MongoDB:ssa on myös resurssien tallettamiselle tarkoitettu mekanismi GridFS, mutta tähän ei tämän tutkimuksen yhteydessä perehdytty (MongoDB, Inc. 2014b).

Koneluettavan rajapinnan kautta voidaan ainoastaan hakea dokumentti sen tunnistetiedon avulla käyttämällä HTTP GET -metodia. Toteuttamatta on vielä POST, PUT ja DELETE -metodit, joilla toteutettaisiin CRUD-toiminnot (Create, Read, Update, Delete). Sovelluksessa rajapinnan ja dojo-asiakasohjelman välinen yhteys noudattaa REST:n periaatteita, mutta dojo-asiakasohjelman ja käyttäjän välinen toiminta ei. Tässä voisi tutkia onko mahdollista muokata Dojo-asiakasohjelmaa siten, että esimerkiksi selaimeen voitaisiin kirjoittamalla siirtyä puuelementin johonkin solmuun. Esimerkiksi siirtää Dojo-sovellus ajoon Node.js-alustalla (Dojo Foundation 2014h).

Tutkimuksen otsikossa luvataan mukautuvaa käyttöliittymää ja tavoitteena oli, että käyttöliittymä pystyisi mukautumaan minkä tahansa muotoiseen JSON-tiedostoon. Tässä on onnistuttu osittain. Käyttöliittymä mukautuu erimuotoisiin JSON-tiedostoihin, mutta mukana tarvitsee olla käyttöliittymän vaatimaa metatietoa. Jos mukana on tuntemattomia avaimia, käyttöliittymä ei osaa käsitellä niitä. Tässä käyttöliittymä voisi tehdä jonkinlaisen oletustoimenpiteen, jos avain ei ole tunnettu esimerkiksi vain näyttää avaimen arvo käyttöliittymässä. Liitteessä 4 on esitelty Object.keys-metodin toimintaa, jonka avulla on mahdollista käydä läpi tuntemattomia avaimia. Näin käyttöliittymä saataisiin mukautumaan myös sellaisiin JSON-tiedostoihin, jotka sisältävät tuntemattomia avaimia.

Käyttöliittymässä on tällä hetkellä tuki vain yhteen elektroniseen prosessioppaaseen kerrallaan. Tuki useammalle prosessioppaalle voitaisiin lisätä esimerkiksi drop down -valikon avulla, joka lataisi sisältönsä JSON-dokumentista. JSON-dokumentissa voisi olla luettelo juuridokumenttien tunnisteista ja tunnistetta vastaavasta nimestä (_id ja name -avaimista).

Tällä hetkellä dokumentit tarvitsee syöttää MongoDB:een manuaalisesti tai esimerkiksi kolmannen osapuolen Robomongo-sovellusta käyttämällä. Dojo-asiakasohjelman käyttöliittymässä voisi olla ominaisuus, jolla se voitaisiin asettaa esimerkiksi sisällöntuottotilaan. Näin voitaisiin esimerkiksi lisätä, päivittää ja poistaa dokumentteja REST:n mukaisesti MongoDB-tietokannasta koneluettavan rajapinnan kautta, olettaen, että rajapintaan on lisätty toimintoja vastaavat operaatiot.

Tällä hetkellä Dojo-sovellus ja Node.js-rajapinta eivät tue minkäänlaista tunnistautumista tai pääsynhallintaa. Nämä olisi hyvä ottaa huomioon esimerkiksi silloin, kun rajapinnalle lisätään dokumenttien lisäys-, päivitys ja poisto-ominaisuudet. Lisäksi jos sovellusta käytetään elektronisena prosessioppaana, yhteydet tulisi suojata, jotta voidaan varmistaa oppaan tietojen oikeellisuus.

Kaiken kaikkiaan tutkimus onnistui, vaikka kaikki työssä käytetyt työkalut ja menetelmät olivat työn tekijälle ennestään tuntemattomia. Pienten ohjelmistoyksiköiden prosessistandardi saatiin analysoitua ja siitä luotua JSON-dokumentteja tietokantaan testidataksi. Valittiin useita JavaScript-pohjaisia vapaasti saatavilla olevia työkaluja työn tekemiseen sekä esiteltiin niiden käyttöä. Näin voitiin toteuttaa yhtä ohjelmointikieltä käyttämällä selain- ja palvelinpään ratkaisut. Otettiin käyttöön MongoDB ja talletettiin sinne dokumentteja. Luotiin koneluettava rajapinta tietokannan dokumenteille Node.js:n ja siihen ladattavien kolmansien osapuolten moduulien avulla. Lisäksi luotiin mukautuva käyttöliittymä Dojo-työkalupakilla, joka käyttää koneluettavaa rajapintaa JSON-dokumenttien hakemiseen, sekä vielä mukautuu ladatun JSON-dokumentin rakenteeseen.

Kohtalaisella jatkokehityksellä koneluettavaan rajapintaan voidaan lisätä dokumenttien lisäys-, muokkaus- ja poisto-operaatiot. Sama koskee myös mukautuvaa käyttöliittymää, johon voitaisiin lisätä kyky lukea tuntemattomia avaimia JSON-dokumentista ja mukautua niihin. Alun perin asetetut vaatimukset mukautuvasta käyttöliittymästä ja koneluettavasta rajapinnasta ovat siis saavutettavissa.

LÄHTEET

Alexandre, S., Mäkinen, T., Varkoi, T. 2008. Implementation of a Software Process Standard as an Electronic Process Guide. SPICE 2008. Proceedings of 8th International SPICE Conference. 26-28 May 2008.

Chodorow, K. 2013. MongoDB: The Definitive Guide, 2. painos. Yhdysvallat, O'Reilly Media, Inc. 409 s.

Coenraets, Christophe. 2012. Creating a REST API using Node.js, Express, and MongoDB. [WWW]. [Viitattu 9.11.2014]. Saatavissa: http://coenraets.org/blog/2012/10/creating-a-rest-api-using-node-js-express-and-mongodb/.

Dojo Foundation. 2014a. Advanced AMD Usage. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.9/modules_advanced/.

Dojo Foundation. 2014b. API Documentation: dijit/_WidgetBase. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/api/1.9/dijit/_WidgetBase.html.

Dojo Foundation. 2014c. API Documentation: dijit/Tree. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/api/1.9/dijit/Tree.html.

Dojo Foundation. 2014d. API Documentation: dojo/store/JsonRest. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/api/1.9/dojo/store/JsonRest.html.

Dojo Foundation. 2014e. Connecting a Store to a Tree. [WWW]. [Viitattu 11.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.6/store_driven_tree/.

Dojo Foundation. 2014f. Creating Template-based Widgets. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.6/templated/.

Dojo Foundation. 2014g. Introduction to AMD Modules. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.9/modules/.

Dojo Foundation. 2014h. Dojo and Node.js. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.8/node/.

Dojo Foundation. 2014i. Dojo Reference Guide: dijit._WidgetBase. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/reference-guide/1.9/dijit/_WidgetBase.html.

Dojo Foundation. 2014j. Dojo Reference Guide: dijit/layout/BorderContainer. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/reference-guide/1.9/dijit/layout/BorderContainer.html.

Dojo Foundation. 2014k. Dojo Reference Guide: dijit/registry. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/reference-guide/1.9/dijit/registry.html.

Dojo Foundation. 2014l. Dojo Reference Guide: dojo/ready. [WWW]. [Viitattu 3.11.2014]. Saatavissa: http://dojotoolkit.org/reference-guide/1.9/dojo/ready.html.

Dojo Foundation. 2014m. Dojo Reference Guide: dojo/store/JsonRest. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/reference-guide/1.9/dojo/store/JsonRest.html.

Dojo Foundation. 2014n. Dojo Reference Guide: Writing Your Own Widget. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://dojotoolkit.org/reference-guide/1.9/quickstart/writingWidgets.html.

ECMA-262. 2011. The ECMAScript Language Specification. 5.1. painos. Geneve: Ecma International. 245 s.

ECMA-404. 2013. The JSON Data Interchange Format. Geneve: Ecma International. 7 s.

Express. 2014. Express Introduction. [WWW]. [Viitattu 17.11.2014]. Saatavissa: http://expressjs.com/.

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Information and Computer Science. Irvine. 162 s.

Haikala, I., Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. 12., uudistettu painos. Hämeenlinna, Talentum Media Oy. 242 s.

ISO/IEC TR 29110-1:2011(E). 2011. Software engineering — Lifecycle profiles for Very Small Entities (VSEs) — Part 1: Overview. Geneve: ISO/IEC. 14 s.

ISO/IEC TR 29110-5-1-1:2012(E). 2012. Software engineering — Lifecycle profiles for Very Small Entities (VSEs) — Part 5-1-1: Management and engineering guide: Generic profile group: Entry profile. Geneve: ISO/IEC. 32 s.

MongoDB, Inc. 2014a. Data Model Examples and Patterns. [WWW]. [Viitattu 9.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/applications/data-models/.

MongoDB, Inc. 2014b. GridFS. [WWW]. [Viitattu 10.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/core/gridfs/.

MongoDB, Inc. 2014c. Model One-to-Many Relationships with Embedded Documents. [WWW]. [Viitattu 9.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/tutorial/model-embedded-one-to-many-relationships-between-documents/.

MongoDB, Inc. 2014d. Model One-to-Many Relationships with Document References. [WWW]. [Viitattu 9.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/tutorial/model-referenced-one-to-many-relationships-between-documents/.

MongoDB, Inc. 2014e. Model One-to-One Relationships with Embedded Documents. [WWW]. [Viitattu 9.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/tutorial/model-embedded-one-to-one-relationships-between-documents/.

MongoDB, Inc. 2014f. Node.js MongoDB Driver. [WWW]. [Viitattu 9.11.2014]. Saatavissa: http://docs.mongodb.org/ecosystem/drivers/node-js/.

Mozilla Developer Network and individual contributors (MDN). 2014. JavaScript Overview. [WWW]. [Viitattu 3.11.2014]. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/JavaScript_Overview.

Mäkinen, T., Varkoi, T. 2008. Analyzing a Process Profile for Very Small Software Enterprises. SPICE 2008. Proceedings of 8th International SPICE Conference. 26-28 May 2008.

Node Packaged Modules. 2014a. NPM: Express. [WWW]. [Viitattu 17.11.2014]. Saatavissa: https://www.npmjs.org/package/express.

Node Packaged Modules. 2014b. NPM: MongoDB Driver. [WWW]. [Viitattu 17.11.2014]. Saatavissa: https://www.npmjs.org/package/mongodb.

RFC 2616. 1999. Hypertext Transfer Protocol HTTP/1.1. The Internet Engineering Task Force. 176 s.

Rodriguez, A. 2008. RESTful Web services: The basics. [WWW]. [Viitattu 1.5.2014]. Saatavissa: http://www.ibm.com/developerworks/webservices/library/ws-restful/.

Russell, M. 2008. Dojo: The Definitive Guide. 1. painos. Yhdysvallat, O'Reilly Media, Inc. 451 s.

Teixeira, P. 2013. Professional Node.js: Building JavaScript-Based Scalable Software. John Wiley & Sons, Inc. 371 s.

LIITTEET

Liite 1. Käyttöönotto

A) MongoDB (Windows)

MongoDB:n asennus/purkaminen

Ladattu MongoDB:n tiedosto ladataan, siirretään ja puretaan C:\ -juureen. Tämän jälkeen voidaan siirtää sisältö `mongodb`-kansioon C:\ -juureen seuraavalla komennolla:

```
cd \
move C:\mongodb-win32-* C:\mongodb
```

MongoDB:n ympäristön asettaminen

MongoDB vaatii datahakemiston, johon se tallettaa kaiken datan. MongoDB:n datahakemiston oletuspolku on `\data\db`. Komentokehotteessa esimerkiksi `C:\` -juureen:

```
md \data
md \data\db
```

Datatiedostoille voidaan määrittää vaihtoehtoinen polku käyttämällä `--dbpath` -optiota mongod.exe -tiedostoon. Esimerkiksi:

```
C:\mongodb\bin\mongod.exe --dbpath C:\data
```

MongoDB:n käynnistäminen

MongoDB voidaan käynnistää ajamalla mongod.exe esimerkiksi komentokehotteessa:

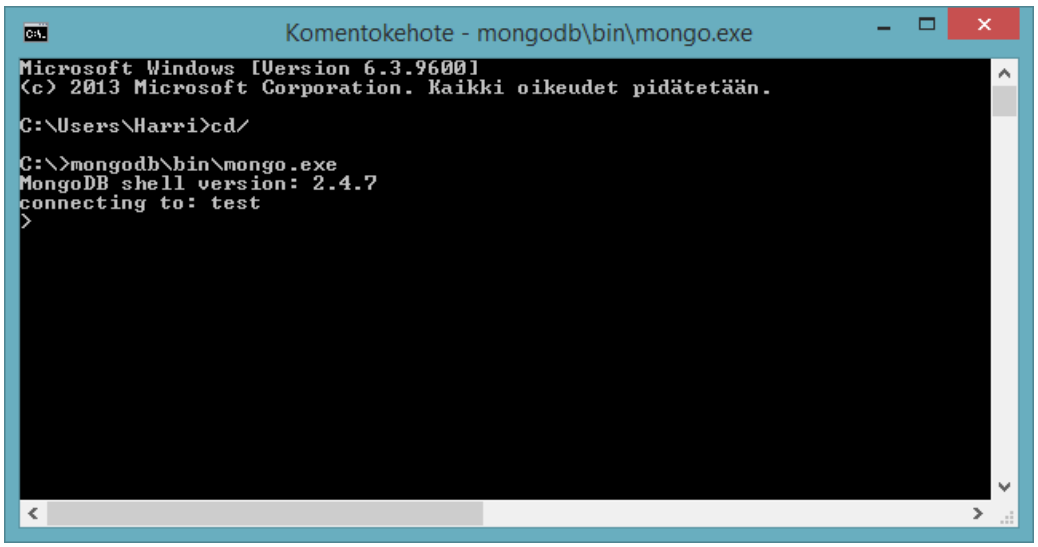
```
C:\mongodb\bin\mongod.exe
```

Huom! MongoDB voi käynnistyä jo vaihtoehtoisen datatiedostojen polun asettamisen aikana.

MongoDB:iin yhdistäminen

Jotta saadaan yhteys `mongo`-shelliin, tarvitaan toinen komentokehote, johon syötetään komento (Kuva 1):

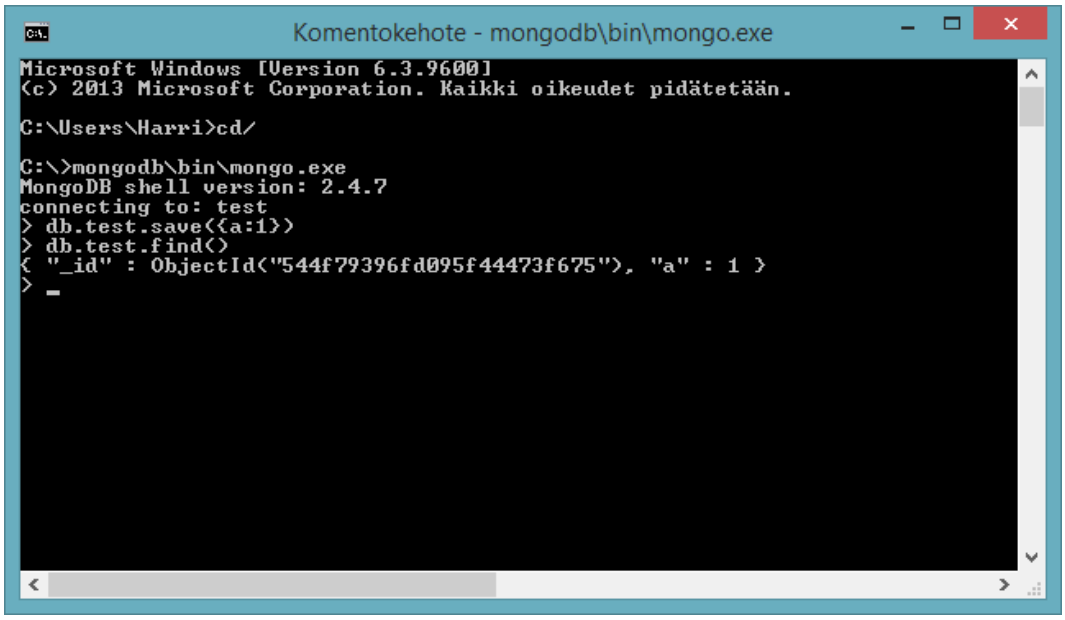
```
C:\mongodb\bin\mongo.exe
```



Kuva 1 - mongo-shelliin on saatu yhteys onnistuneesti

Oletuksena mongod.exe -tiedostoa ajetaan `localhost`in portissa `27017`, `mongo`-shell ottaa tähän yhteyden. Tietueen lisäämistä voidaan testata kirjoittamalla `mongo`-shelliin seuraavat komennot, joilla lisätään dokumentti `test`-kokoelmaan, joka on `test`-oletustietokannassa. Tämän jälkeen dokumentti haetaan (Kuva 2):

```
db.test.save( { a: 1 } )
db.test.find()
```



Kuva 2 - Tietue on lisätty onnistuneesti `test`-tietokannassa olevaan `test`-kokoelmaan

Kannasta palautuu seuraavaa:

```
{
    “_id” : “ObjectId("544f79396fd095f44473f675")”,
    “a” : 1
}
```

Kannassa siis on sinne syötetty arvo-avain -pari ja tämän lisäksi MongoDB on generoinut avain-arvo -parin yhteyteen oman `_id` -tunnisteensa.

Samalla huomataan, että testi on onnistunut ja MongoDB on toimintakunnossa.

Sovelluksen käyttämä tietokantamateriaali

Kun MongoDB:n ympäristö on kunnossa, voidaan tuoda (import) `mongoimport`-komennolla .json-tiedosto, joka sisältää tietyn kokoelman ja kokoelman dokumentit. Esimerkiksi `vsetest.json`-tiedoston tuominen `test`-tietokannan `vsetest`-kokoelmaan onnistuu, kun siirretään `vsetest.json`-tiedosto kansioon `C:\mongodb\bin` ja kirjoitetaan seuraava komento `mongo`-shelliin:

```
mongoimport --db test --collection vsetest --file vsetest.json
```

Jos MongoDB:ssa jo on joitain dokumentteja `vsetest`-kokoelmassa, `mongoimport`-komentoon voidaan lisätä `--upsert` -optio. Tämä muuttaa tuontiprosessia siten, että tietokannassa jo olemassa olevat objektit päivitetään ja kaikki muut tuotavat objektit lisätään. Tuonti toimii oletuksena `_id` -kentän pohjalta.

Kun tietokantaan on tuotu `vsetest`-kokoelma ja sen dokumentit, websovellukseen liittyvä Node.js-palvelin voidaan ajaa.

Kokoelmien tietokannasta pois vieminen (export) puolestaan on mahdollista `mongoexport`-komennolla `mongo`-shelliin:

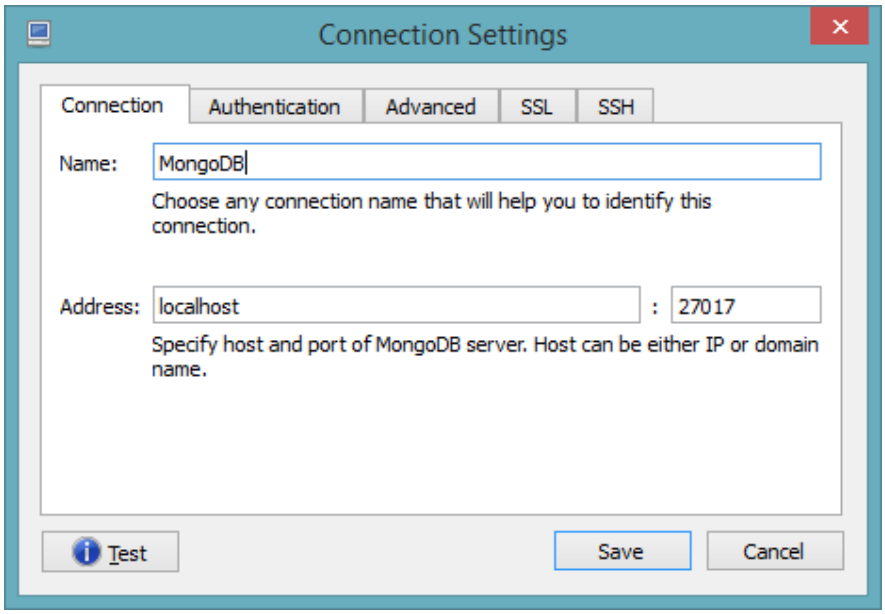
```
mongoexport --collection vsetest --out vsetest.json
```

Komento luo tiedoston `C:\mongodb\bin` -hakemiston tiedoston `vsetest.json`, joka sisältää kokoelman `vsetest` ja kaikki sen sisältämät dokumentit.

Robomongo

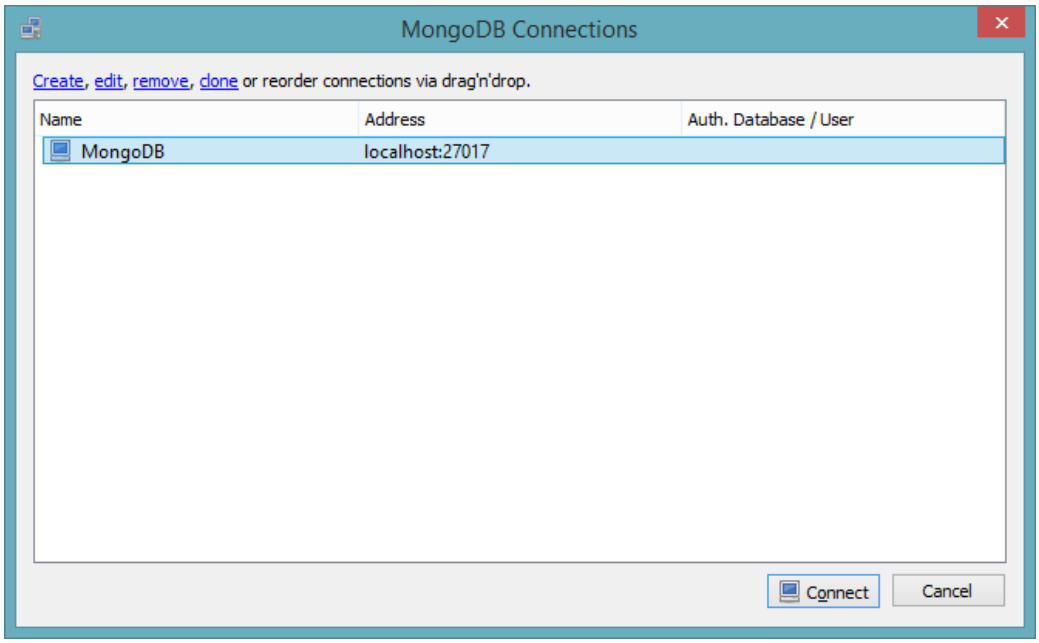
Robomongo on shell-keskeinen avoimen lähdekoodin ja monella alustalla toimiva MongoDB:n hallintatyökalu eli palvelimen ylläpitäjälle graafinen käyttöliittymä. Robomongo sisältää saman JavaScript-moottorin, joka on MongoDB:n `mongo`-shellissä .Eli kaikki, mitä voidaan kirjoittaa `mongo`-shelliin (`mongo.exe`), voidaan kirjoittaa myös Robomongoon.

Robomongon asennuksen jälkeen ohjelma käynnistetään, tämän jälkeen tarvitsee luoda yhteys MongoDB:iin.



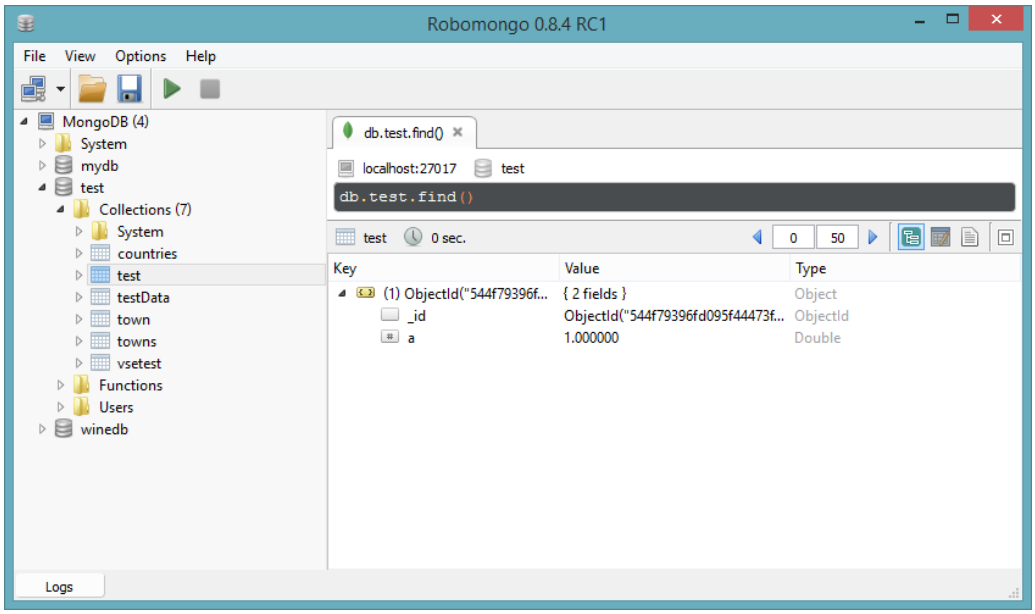
Kuva 3 - Yhteyden asetukset

Robomongon yhteysasetuksista löytyy mm. yhteyden todennukseen ja suojaamiseen liittyviä vaihtoehtoja. Nämä optiot vaativat, että yhteyden todennus ja suojaus on otettu käyttöön MongoDB:n puolelta.



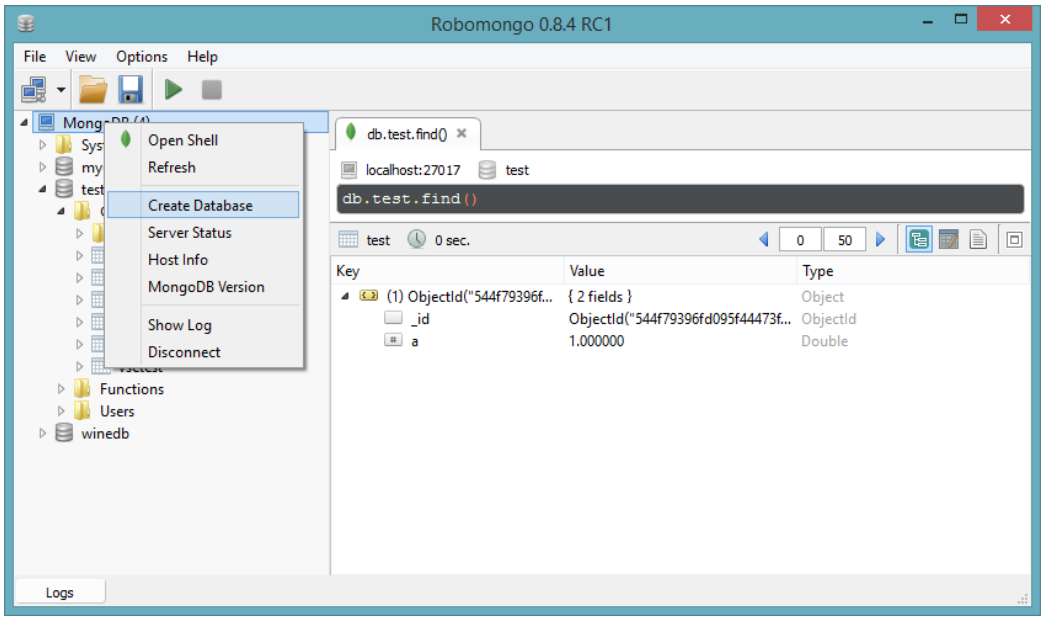
Kuva 4 - Yhteyden asetukset on talletettu

Robomongoon voidaan tallettaa useita yhteysasetuksia.



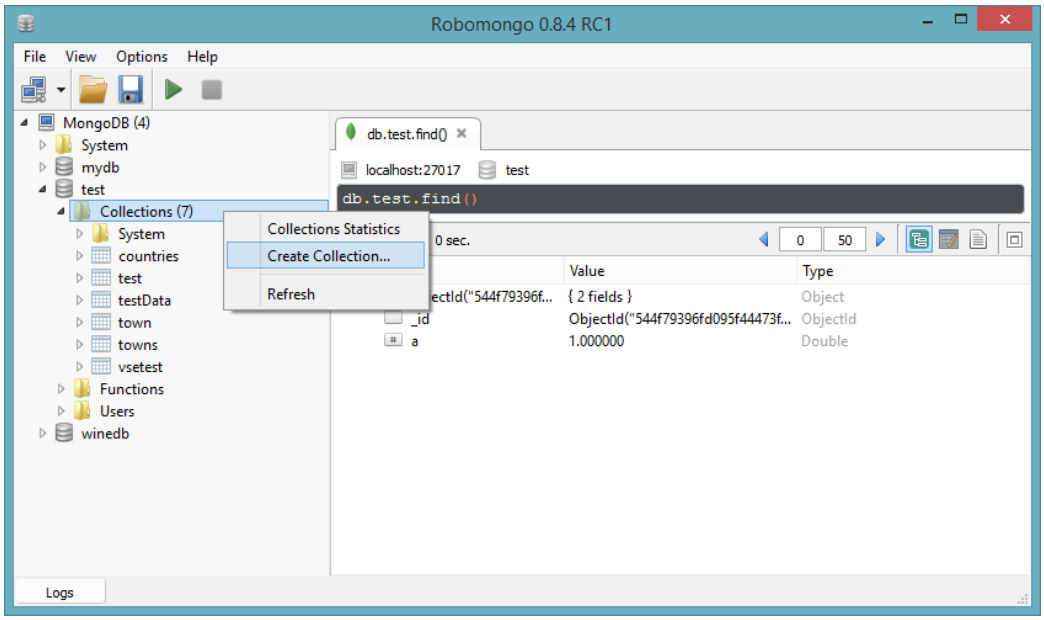
Kuva 5 - Robomongon käyttöliittymä

Robomongon graafisesta käyttöliittymästä (Kuva 5) nähdään mm. MongoDB:ssa olevat tietokannat, niiden alla olevat kokoelmat, sekä kokoelmissa olevat dokumentit. Graafisessa käyttöliittymässä on myös komentorivi, jonka kautta `mongo`-shelliin voidaan antaa komentoja.



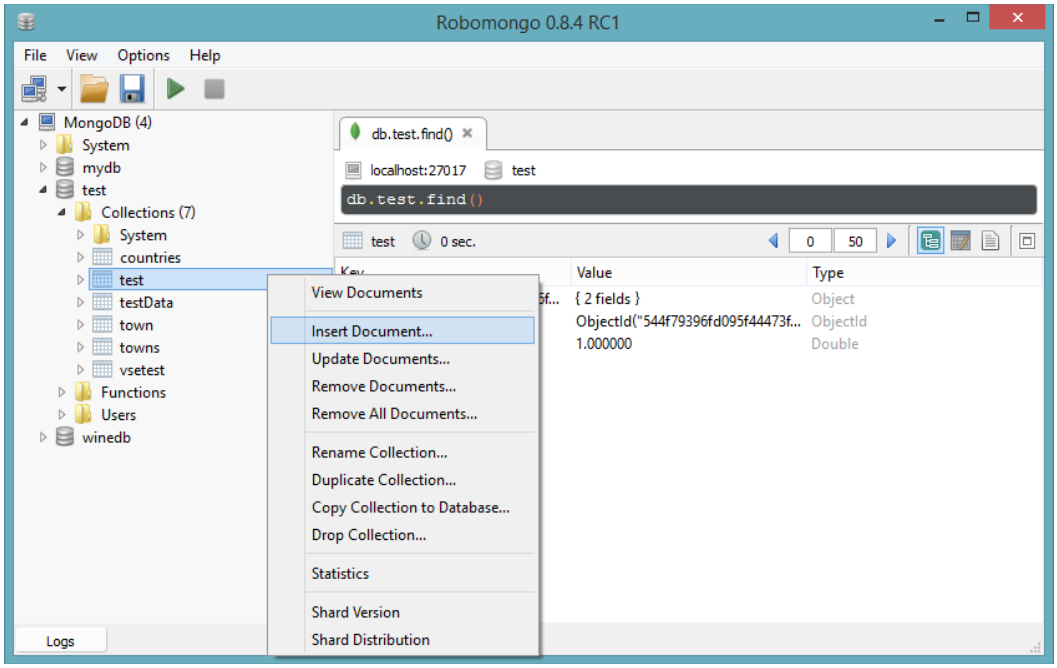
Kuva 6 - Tietokannan luonti MongoDB:iin Robomongon käyttöliittymän kautta

Robomongon graafisen käyttöliittymän kautta voidaan luoda uusi tietokanta (Kuva 6) MongoDB:iin hiiren oikeanpuoleisen painikkeen alta ilmestyvän valikon kautta.



Kuva 7 - Kokoelman lisääminen tietokantaan Robomongon kautta

Kuvassa 7 näkyy kokoelman luominen ja lisääminen tietokannan alle `Collections`-kansion kohdalla hiiren oikeanpuoleisella painikkeella aukeaa valikko, josta löytyy `Create Collection` -vaihtoehto.



Kuva 8 - Dokumentin lisääminen kokoelmaan Robomongon kautta

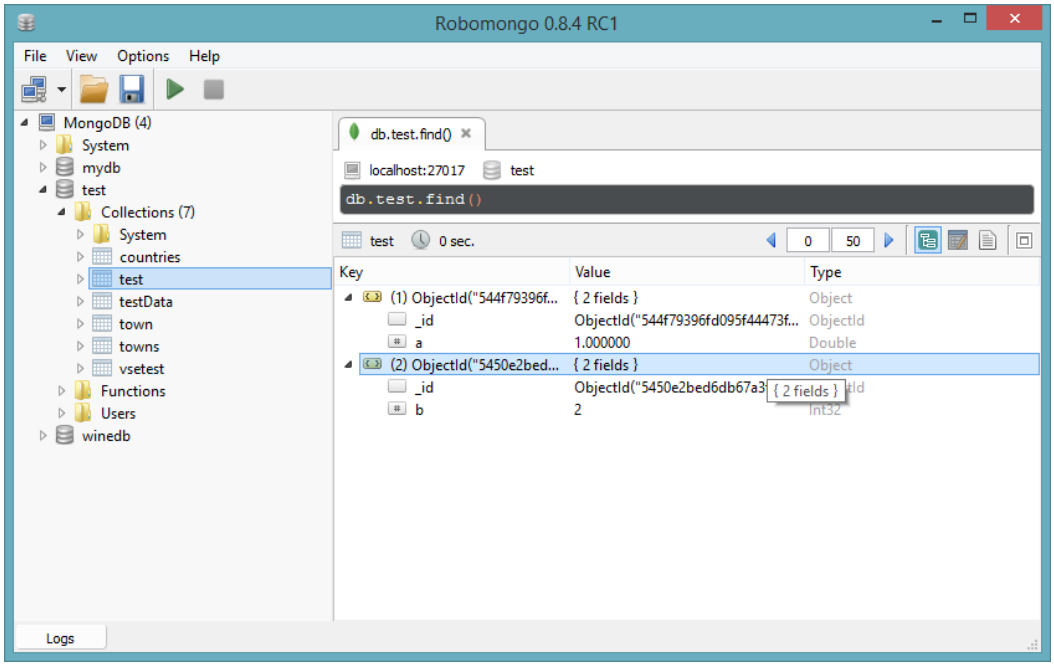
Dokumentin lisääminen onnistuu niin ikään hiiren oikeanpuoleisen painikkeen avulla. Dokumentin lisääminen on havainnollistettu kuvissa 8 ja 9.

Kuva 9 - Dokumentin lisääminen

Kuvan 9 esimerkissä `test`-tietokannassa olevaan `test`-kokoelmaan lisätään dokumentti Robomongon dokumentinlisäystyökalun avulla:

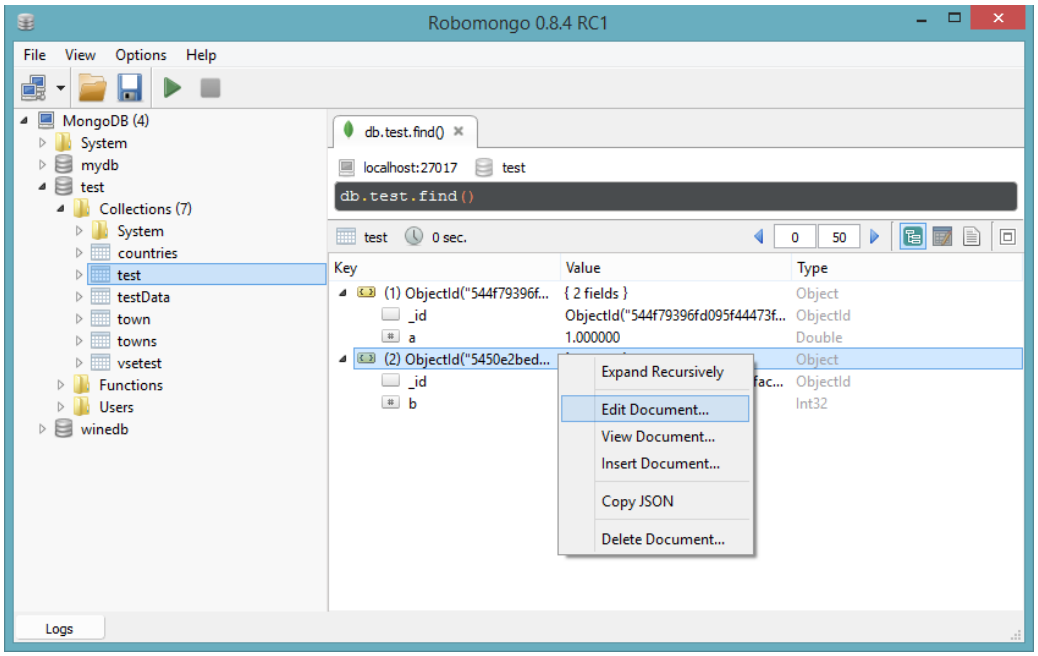
```
{ "b" : 2 }
```

Lisäystyökalusta löytyy myös dokumentin `Validate`-painike, jonka avulla voidaan tarkastaa dokumentin syntaksin oikeellisuus.



Kuva 10 - Uusi dokumentti on lisätty `test`-kokoelmaan

Kuvasta 10 nähdään, että `test`-kokoelmaan on ilmestynyt lisätty dokumentti. MongoDB on generoinut lisättyyn dokumenttiin `_id`-tunnisteen, koska sitä ei manuaalisesti ole dokumenttiin lisätty, sekä sille arvonparin.



Kuva 11 - Dokumenttien muokkausmahdollisuudet

Kuvassa 11nähdään Robomongon dokumentin manipulointi mahdollisuudet. Dokumentteja voidaan siis editoida, katsella tai lisätä hiiren oikeanpuoleisella painikkeella aukeavan valikon avulla. Vaihtoehdoista aukeaa kuvan 11 kaltainen ikkuna. Robomongo mahdollistaa dokumentin (JSONin) kopioinnin suoraan leikepöydälle.

MongoDB Windows-palveluna

MongoDB voidaan asettaa Windows-palveluksi, jolloin tietokanta käynnistyy automaattisesti jokaisen uudelleen käynnistyksen yhteydessä.

1) **Konfiguroi hakemistot ja tiedostot**

   Luodaan konfiguraatiotiedosto ja hakemistopolku MongoDB:n lokin ulostulolle (`logpath`).

   Luodaan hakemisto MongoDB:n lokitiedostoille komennolla:

   ```
   md C:\mongodb\log
   ```

Seuraavaksi luodaan konfiguraatiotiedosto MongoDB:n `logpath`-optiolle:

```
echo logpath=C:\mongodb\log\mongo.log > "C:\mongodb\mongod.cfg"
```

2) MongoDB-palvelun asennus ja ajaminen

Seuraavat komennot on ajettava järjestelmänvalvojan oikeuksilla.

Jotta MongoDB -palvelun asentaminen onnistuisi `--install` -optiota käyttämällä, on määritettävä ajonaikainen optio `logpath`.

```
"C:\mongodb\bin\mongod.exe" --config
"C:\mongodb\mongod.cfg" --install
```

Jos `dbpath`-hakemistoa ei ole, mongod.exe ei käynnisty. Oletusarvo `dbpath`ille on `\data\db`.

MongoDB-palvelu käynnistyy komennolla:
`net start MongoDB`

3) MongoDB-palvelun pysäyttäminen ja poistaminen

Palvelu voidaan pysäyttää komennolla:
`net stop MongoDB`

Windows-palvelu voidaan poistaa komennolla:
`C:\mongodb\bin\mongod.exe --remove`

B) Node.js

Node.js:n asennus

Node.js:n asennuspaketti löytyy osoitteesta http://nodejs.org/. Asennuksen jälkeen voidaan testata sen toimivuus komennolla:

```
node -v
```

Vastauksena Node.js palauttaa versionumeronsa. Node.js siis lisää itsensä Windowsin `PATH`iin, jonka ansiosta voidaan ajaa Node-komentoja komentokehotteella työhakemistosta riippumatta.

Sovelluksen tarvitsemat moduulit

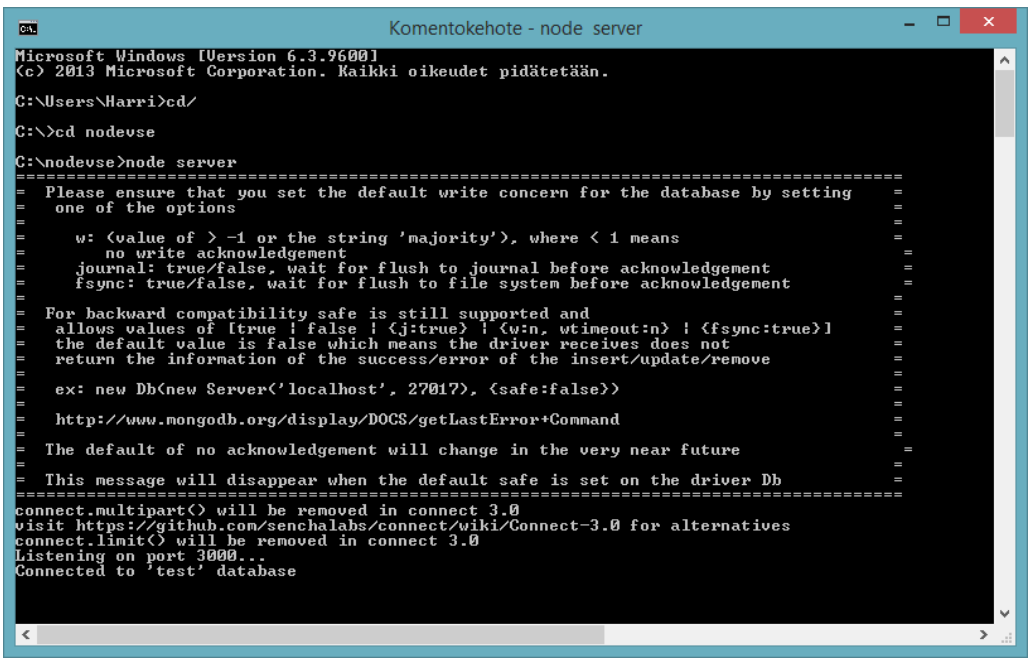
Jos projektikansion mukana ei ole `node_modules` -hakemistoa, tarvitsee sovelluksen tarvitsemat moduulit Express ja MongoDB Node.js Driver -moduulit ladata erikseen NPM:stä. Tämä onnistuu yksinkertaisesti menemällä työhakemistoon komentokehotteella ja syöttämällä komennot:

```
npm install express
npm install mongodb
```

Nämä komennot luovat työhakemistoon node_modules -hakemiston, jonne on kummatkin moduulit ovat latautuneet omiin alihakemistoihinsa.

Node.js:n käynnistys

Web-sovelluksen Node.js -palvelinosuus vaatii käynnistyäkseen, että MongoDB on käynnissä ja sieltä löytyy `test`-tietokanta, jossa puolestaan on `vsetest`-kokoelma.



Kuva 12 - Käynnistetty Node-palvelin

Siirrytään Node -projektin kansioon, joka sijaitsee esim. `C:\`-juuressa ja Node-palvelin käynnistetään komentokehotteen komennolla (Kuva 12):

```
node server
```

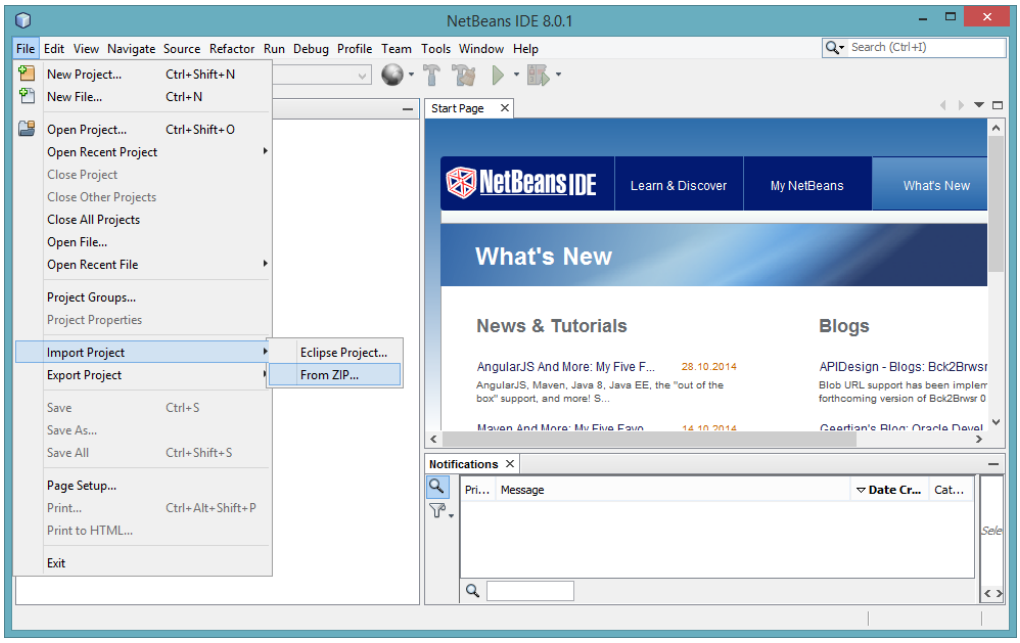
Komennon `server` viittaa tässä Node.js-projektikansiossa olevaan `server.js` -tiedostoon.

Node-palvelimen sammuttaminen onnistuu näppäinyhdistelmällä `Ctrl+C`.

C) Web-sovellus

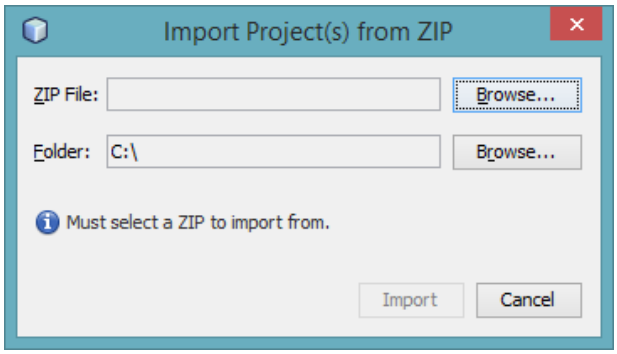
Tuonti (import)

Tuodaan (import) projekti Netbeansiin kuvan 13 mukaisesti Netbeansin sisäisen työkalun avulla ZIP-tiedostosta.



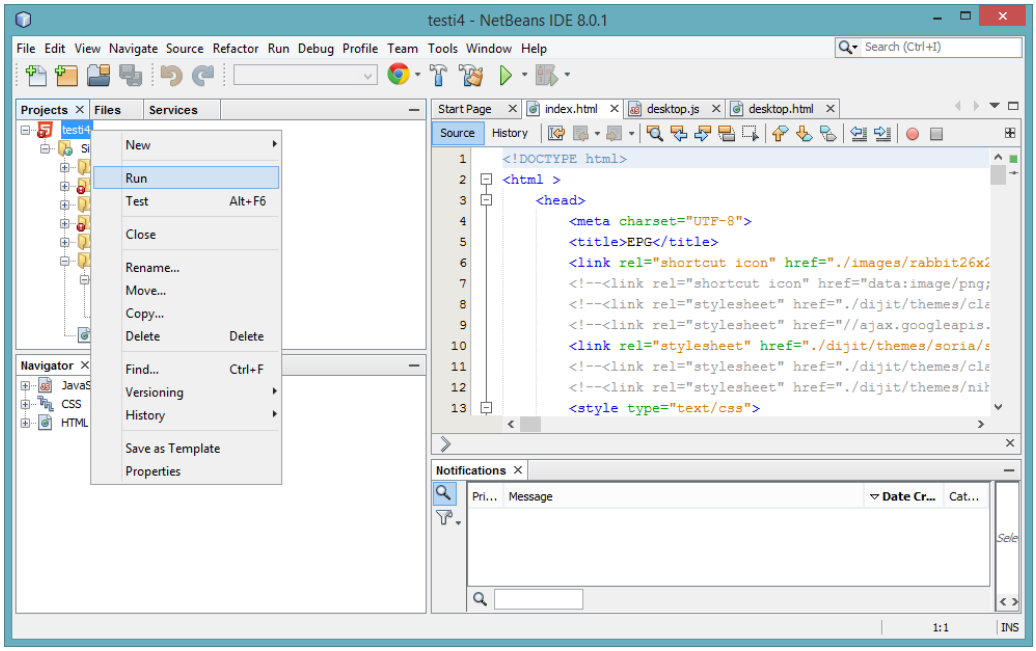
Kuva 13 - Projektin tuonti Netbeansiin

Seuraavaksi aukeaa kuvan 14 mukainen ikkuna, jolla valitaan tuotava projekti. Import-nappia painamalla haluttu projekti tuodaan Netbeans-ympäristöön, kun se on valittu.



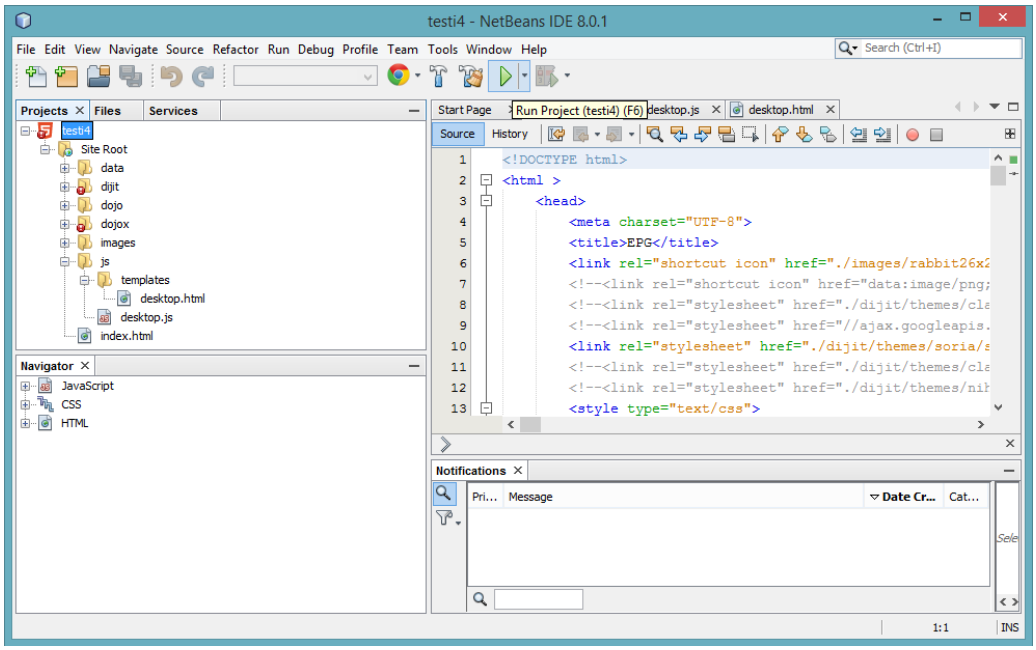
Kuva 14 - Projektin tuonti

Web-sovelluksen onnistunut käynnistyminen vaatii, että MongoDB:iin on talletettu `vse-tree` -dokumentti `vsetest`-kokoelmassa, joka puolestaan on talletettu `test`-tietokantaan.



Kuva 15 - Netbeans-projektin ajaminen

Valitaan haluttu projekti Netbeansin projektinäkymästä näytön vasemmasta sivusta ja tämän jälkeen painetaan hiiren oikealla painikkeella haluttua projektia ja valitaan aukeavasta valikosta `Run` -vaihtoehto (Kuva 15).

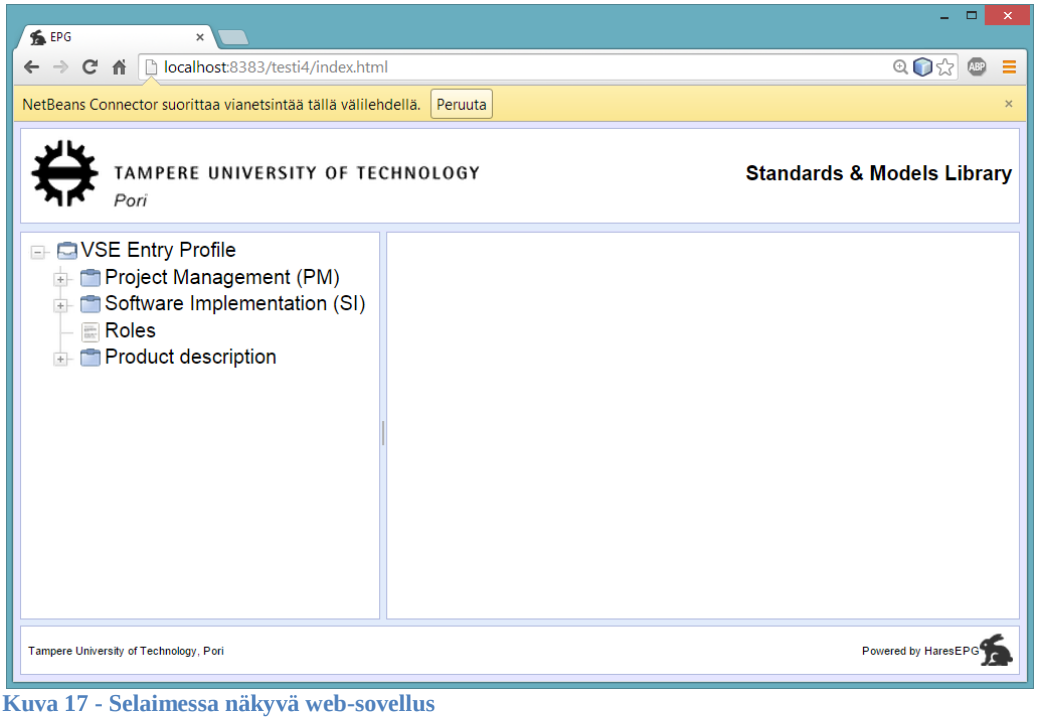


Kuva 16 - Netbeans-projektin ajaminen vaihtoehtoisella tavalla

Vaihtoehtoisesti haluttu projekti voidaan ajaa valitsemalla se vasemmalta ja tämän jälkeen painaa F6-näppäintä tai työkalupalkista näytön yläosasta Run Project -näppäintä (Kuva 16).

D) Sovellus käynnissä

Onnistuneesti käynnistynyt sovellus näyttää kuvan 17 mukaiselta web-selaimessa (Google Chrome):



Kuva 17 - Selaimessa näkyvä web-sovellus

Sovelluksen toimivuus siis vaatii, että ensin on käynnistetty MongoDB-palvelin, tämän jälkeen käynnistetään Node.js -palvelin ja lopuksi ajetaan projekti Netbeansissa.

Liitteen 1 lähteet

Joyent, Inc. 2014. Node.js. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://nodejs.org/.

MongoDB, Inc. 2014. Install MongoDB on Windows. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/tutorial/install-mongodb-on-windows/.

MongoDB, Inc. 2014. Import and Export MongoDB Data. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/core/import-export/.

MongoDB, Inc. 2014. Reference: mongoimport. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://docs.mongodb.org/v2.4/reference/program/mongoimport/.

Oracle Corporation. 2013. Netbeans IDE. [WWW]. [Viitattu 5.11.2014]. Saatavissa: https://netbeans.org/.

Paralect. 2014. Robomongo. [WWW]. [Viitattu 5.11.2014]. Saatavissa: http://robomongo.org/.

Liite 2. MongoDB:n ominaisuuksia ja tietotyyppejä

Alikokoelmat

Yksi käytäntö kokoelmien järjestämiselle on käyttää nimiavaruuden mukaisia alikokoelmia pisteellä (.) erotettuna. Esimerkiksi, sovellus, joka sisältää blogin voisi omistaa kokoelman `blog.posts` sekä erillisen kokoelman `blog.authors`. Tämä on vain järjestämistä varten, mitään suhdetta `blog`-kokoelman kanssa ja sen lapsielementtien kanssa ei ole (`blog`-kokoelman ei edes tarvitse olla olemassa). Vaikka alikokoelmilla ei ole mitään erikoisia ominaisuuksia, ne ovat käytännöllisiä ja sisällytettyinä useisiin MongoDB:n työkaluihin:

GridFS on protokolla suurien tiedostojen tallettamiselle ja käyttää alikokoelmia tiedoston metadatan tallettamiselle sisältöpalasista eriytettynä.

Suurin osa ajureista tarjoaa syntaktista helpotusta annetun kokoelman alikokoelman käyttöön. Esimerkiksi, tietokannan shellissä `bd.blog` -komento antaa `blog`-kokoelman ja `db.blog.posts` antaa `blog.posts`-kokoelman.

Alikokoelmat on hyvä tapa järjestää dataa MongoDB:ssa ja niiden käyttöä suositellaan. Ketjuttamalla tietokannan nimen kannassa olevan kokoelman kanssa saadaan täysin pätevä kokoelman nimi, jota kutsutaan nimiavaruudeksi (namespace). Esimerkiksi, jos käytetään `blog.posts`-kokoelmaa `cms`-tietokannassa, kokoelman nimiavaruus olisi `cms.blogs.posts`. Nimiavaruudet ovat rajoitettu 121 tavun pituuteen ja käytännössä niiden tulisi olla alle 100 tavua pitkiä.

Päivämäärät

JavaScriptissä `Date`-luokkaa käytetään MongoDB:n päivämäärätyyppiin. Kun luodaan uusi `Date`-objekti, kutsutaan aina `new Date(…)`, ei ainoastaan `Date(…)`. Kutsuttaessa muodostinta funktiona (eli ilman `new`:tä) palauttaa merkkijono esityksen päivämäärästä, eikä varsinaista `Date`-objektia. Tämä ei johdu MongoDB:stä, vaan tämä kuuluu JavaScriptin toimintaperiaatteisiin. Jos ei aina käytetä `Date`-luokan muodostinta, voidaan saada aikaan merkkijonojen ja päivämäärien sotku. Merkkijonot ja päivämäärät eivät ole yhteensopivia keskenään ja tämä voi aiheuttaa ongelmia poistamisen, päivittämisen, kyselyjen ja aikalailla kaiken kanssa.

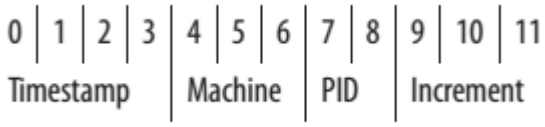
Shellissä olevat päivämäärät näytetään käyttämällä paikallista aikavyöhykettä. Kuitenkin, tietokanassa olevat päivämäärät on talletettu epookki-millisekunteina, joten niihin

ei ole yhdistetty aikavyöhyketietoja. (Aikavyöhyketieto voitaisiin tallettaa joksikin avain/arvo-pariksi.)

ObjectID

ObjectId käyttää 12 tavua tallennustilaa, jolla saadaan 24 heksadesimaalilukua pitkä merkkijono: kaksi numeroa jokaista tavua kohden. ObjectId esitetään usein pitkällä heksadesimaalimerkkijonolla, mutta merkkijono on itse asiassa kaksi kertaa niin suuri kuin talletettava data.

Jos luo useita ObjectId:tä pienellä aikavälillä, huomataan, että ainoastaan lopussa olevat numerot muuttuvat. Lisäksi muutama ObjectId:n keskivaiheilla olevaa numeroa muuttuvat (jos viivästyttää luomista muutamalla sekunnilla). Tämä johtuu tavasta, jolla ObjectId:tä luodaan. ObjectId:n 12 tavua syntyy seuraavasti:



ObjectId:n ensimmäiset neljä tavua ovat aikaleima sekunteina epookista. Tämä tarjoaa muutaman hyödyllisen ominaisuuden:

- Kun yhdistetään aikaleima viidellä seuraavalla tavulla, se tarjoaa ainutlaatuisuutta sekunnin sisällä

- Koska aikaleima on järjestyksessä ensimmäinen, tarkoittaa se sitä, että ObjectId:t lajitellaan karkeasti lisäysjärjestykseen. Tämä ei takaa mitään, mutta saa esimerkiksi ObjectId:t tehokkaasti indeksoitumaan.

- Näissä neljässä tavussa piilee implisiittinen aikaleima siitä, milloin dokumentti on luotu. Useimmat ajurit tarjoavat metodin tämän tiedon talteen ottamiselle ObjectId:stä.

- Koska käytetään nykyistä aikaa, jotkut käyttäjät voivat olla huolissaan, että heidän palvelimiensa kellot täytyy synkronisoida. Vaikka synkronisoidut kellot ovat hyvä idea, varsinaisella aikaleimalla ei ole väliä ObjectId:lle, vain että se uudistuu usein ja kasvaa.

- ObjectId:n seuraavat kolme tavua ovat yksilöllinen tunniste laitteelle, jossa se luotiin. Tämä on yleensä hash laitteen isäntänimestä. Näillä varmistetaan, ettei luoda samoja ObjectId:tä.

- Seuraavat kaksi tavua saadaan ObjectId:tä luovan prosessin prosessitunnisteesta (PID), jotta saadaan yksilöllisyyttä samalla laitteella samanaikaisesti ObjectId:tä luoviin eri prosesseihin.

- ObjectId:n yhdeksän ensimmäistä tavua varmistavat sen yksilöllisyyden useiden laitteiden ja prosessien osalta yhden sekunnin aikavälin sisällä. Viimeiset kolme tavua ovat yksinkertaisesti vain kasvava laskuri, joka on vastuussa ainutlaatuisuudesta sekunnin sisällä yhdessä prosessissa. Tämä mahdollistaa 256^3 (16,777,215) ainutlaatuista ObjectId:tä luotavaksi yhden sekunnin aikana yhdessä prosessissa.

Liitteen 2 lähteet

Chodorow, K. 2013. MongoDB: The Definitive Guide, 2. painos. Yhdysvallat, O'Reilly Media, Inc. 409 s.

Liite 3. Dojo toolkitin termistöä

Työkalupakki
Työkalupakki on nimensä mukaisesti kokoelma työkaluja. Työkalupakkeja käytetään ohjelmoinnin puitteissa käyttöliittymäsuunnittelussa. Dojon tarkin määritelmä on työkalupakki, koska se on enemmän kuin kirjastollinen tukevaa koodia, joka tarjoaa joukon siihen liittyviä funktioita ja abstraktioita; se tarjoaa myös käyttöönoton apuvälineitä, testaustyökaluja, sekä paketointijärjestelmän. On helppo päätyä kiistelemään kirjasto vastaan ohjelmistokehys vastaan työkalupakki, mutta Dojo on ristitty työkalupakiksi.

Moduuli
Fyysisesti Dojon moduuli ei ole muuta kuin JavaScript-tiedosto tai hakemistollinen yhteenkuuluvia JavaScript-tiedostoja. Ylätason hakemisto myös määrittelee sinne kuuluvan koodin nimiavaruuden. Dojon moduulit ovat verrattavissa muiden ohjelmointikielien paketteihin, sillä niitä käytetään lokeroimaan toisiinsa liittyvät ohjelmistokomponentit.

Resusrssi
Kun Dojo-moduuli jaetaan useaan tiedostoon tai moduuli koostuu vain yhdestä JavaScript-tiedostosta, ovat nämä tiedostot silloin lähteitä. Vaikkakin resurssia voitaisiin ainoastaan käyttää loogisesti organisoimaan erilaisia moduuliin liitettyjä abstraktioita, pietään myös mielessä JavaScript-tiedoston koon pienentäminen. Kompromissi vastaa oleellisilta osiltaan tiedostokokojen pienentämistä, ettei tarvitse ladata tarpeetonta koodia, samalla myös lataamatta liian monta pientä tiedostoa, jotka kaikki ovat synkronisia kutsuja ja aiheuttavat tehoresurssien kulumista kommunikoimalla takaisin web-palvelimelle.

Nimiavaruus
Fyysisesti Dojon nimiavaruudet kartoittavat saman tiedostojärjestelmänhierarkian, kuin joka määrittelee moduulit ja resurssit; nimiavaruus estää samalla tavalla nimettyjen moduulien ja resurssien ristiriitoja. Dojo suojaa sivun globaalia nimiavaruutta, ja mikä tahansa Dojolla luotu moduuli ei saastuta globaalia nimiavaruutta, jos se on toteutettu kunnolla. On muistettava, että kaikki Basessa oleva sopii ylimmän tason dojo-nimiavaruuteen.

Funktio
Funktio on koodipalanen, joka on määritelty kerran, mutta voidaan ajaa useaan kertaan. JavaScriptissa funktiot ovat olioita, joita voidaan kierrättää kuten mitä tahansa muuttujaa. Muodostinfunktio on funktio, jota käytetään erityisesti uuden operaattorin kautta, joka luo uuden JavaScript-funktio -olion ja suorittaa sen alustamisen. Kaikki JavaScript-oliot perivät JavaScriptin sisäänrakennetusta oliotyypistä ja niillä on prototyyppiominaisuus, joka johtuu JavaScriptiin periytymismekanismista, joka puolestaan perustuu pro-

totyyppiketjutukseen. Dojossa termi muodostin voi myös viitata anonyymiin funktioon, joka kartoittaa muodostinavaimen dojo.declaren assosiaatiotaulukkoon ja jota pääsääntöisesti käytetään Dojo-luokan ominaisuuksien alustamiseen.

Olio
Olion yleiskäsite viittaa JavaScriptissa yhdistedatatyyppiin, joka voi sisältää vaikka kuinka paljon nimettyjä ominaisuuksia. Esimerkiksi, yksinkertainen lause var o={} käyttää olion kirjaimellista syntaksia luomaan JavaScript-olion. Termiä ”assosiatiivinen taulukko” käytetään joskus kuvaamaan avain/arvo-pareja, kuten {a:1,b:2}, sen sijaan, että niitä kutsuttaisiin olioiksi. Teknisesti puhuen, JavaScriptillä on ainoastaan olioita eikä luokkia, vaikkakin Dojo simuloi käsitystä luokasta dojo.declare-funktion kautta, jota käytetään erityisesti tähän tarkoitukseen.

Ominaisuus
Olio-ohjelmoinnissa mitä tahansa datan palasta, joka on talletettu luokkaan, kutsutaan ominaisuudeksi. Dojon tapauksessa, tämä voi viitata dataan, joka sisältyy funktio-olioihin tai dataan, joka sisältyy dojo.declaressa määriteltyihin Dojo-luokkiin.

Metodi
Olio-ohjelmoinnin kontekstissa, sisältäen JavaScriptin sekä Dojon, luokan jäsenenä olevaa funktiota kutsutaan yleensä metodiksi. Lisäksi Dojossa anonyymit funktiot, jota esiintyvät dojo.declaressa ovat myös metodeja, koska dojo.declare tarjoaa lähtökohdat luokkapohjaiselle periytymismekanismille.

Luokka
Esittely (declaration), joka edustaa loogista oliota, joka on määritelty dojo.declare -funktion avulla (funktio, joka on erityisesti suunniteltu simuloimaan luokkia ja periytymishierarkioita) on luokka. Termiä käytetään löyhästi, koska JavaScript ei tue luokkia samassa merkityksessä, kuin ne ovat kielissä, kuten Java tai C++.

Pienoisohjelma
Dojon pienoisohjelma on dojo.declare-lauseella luotava funktio-olio, joka sisältää dijit._Widgetin isäntäelementtinä. Yleensä pienoisohjelmalla on näkyvä olomuoto ruudulla ja se niputtaa loogisesti HTML-, CSS-, JavaScript-, ja staattiset resurssit yhdistyneeksi olioksi, jota voidaan manipuloida, ylläpitää ja siirtää aivan kuin tiedostoa.

Define ja require -kutsut

`Define` -kutsua käytetään moduulien määrittelyssä. `Define`-kutsua käytetään tavallisesti JavaScript-tiedostossa. JavaScript-tiedosto määrittää moduulin. Kaikki Dojo-tiedostot käyttävät `define`-kutsua. (Dojo Foundation 2014a, 2014g)

`Require`-kutsua käytetään, kun tarvitaan moduuleja, jotka on jo määritelty. `Require`-kutsua käytetään tavallisesti HTML-sivuilla. HTML-sivu ei ole moduuli, mutta se vaatii moduuleja, että sivu voidaan näyttää käyttäjälle. (Dojo Foundation 2014a, 2014g)

Liitteen 3 lähteet

Dojo Foundation. 2014a. Advanced AMD Usage. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.9/modules_advanced/.

Dojo Foundation. 2014g. Introduction to AMD Modules. [WWW]. [Viitattu 4.11.2014]. Saatavissa: http://dojotoolkit.org/documentation/tutorials/1.9/modules/.

Russell, M. 2008. Dojo: The Definitive Guide. 1. painos. Yhdysvallat, O'Reilly Media, Inc. 451 s.

Liite 4. Tuntemattomat avaimet JSON-dokumentissa

Listauksessa 1 on esitelty koodipätkä, jolla voidaan käydä läpi olion ennalta tuntemattomat ominaisuuksien nimet. Tässä on käytetty `Object.keys`-metodia, joka palauttaa taulukon annetun olion numeroituvista ominaisuuksista. Toiminnallisuus oli tarkoitus liittää Dojo web-sovelluksen itse rakennetun `Desktop`-pienoisohjelman `onClick`-metodin toiminnallisuuteen.

Listaus 1 - Tuntemattomien ominaisuuksien nimet

```
onClick: function(item) {
    for (var i = 0; i < item.items.length; i++) {

        var group = item.items[i];
        var allPropertyNames = Object.keys(group);

        for (var j=0; j<allPropertyNames.length; j++) {
            var name = allPropertyNames[j];
            var value = group[name];
            // Konsoliin ominaisuuden nimi
            console.log(name);
            //console.log(value);
        }
    }
}
```

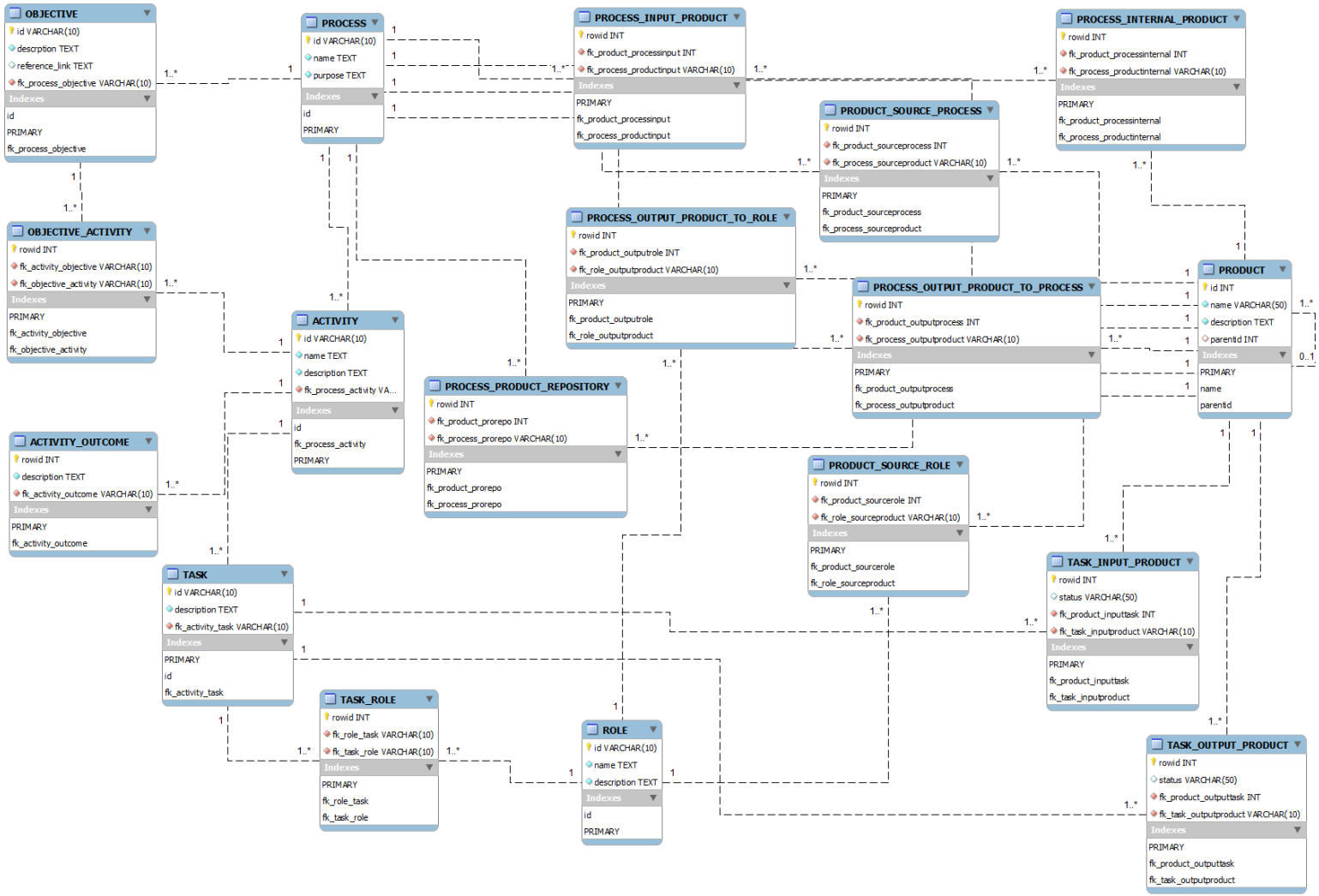
Listauksen 1 toiminnallisuus, jolla etsitään olion kehittäjälle ennestään tuntemattomat ominaisuudet. Tämä olisi jatkokehityksen kannalta mielenkiintoista, koska tämän toiminnallisuuden avulla Dojo web-sovellus voisi niin sanotusti ottaa sisään minkä muotoisia JSON-dokumentteja tahansa, ilman että kehittäjän tarvitsisi ennakkoon tietää JSON-dokumentin sisältämien avaimien nimet.

Lähteet

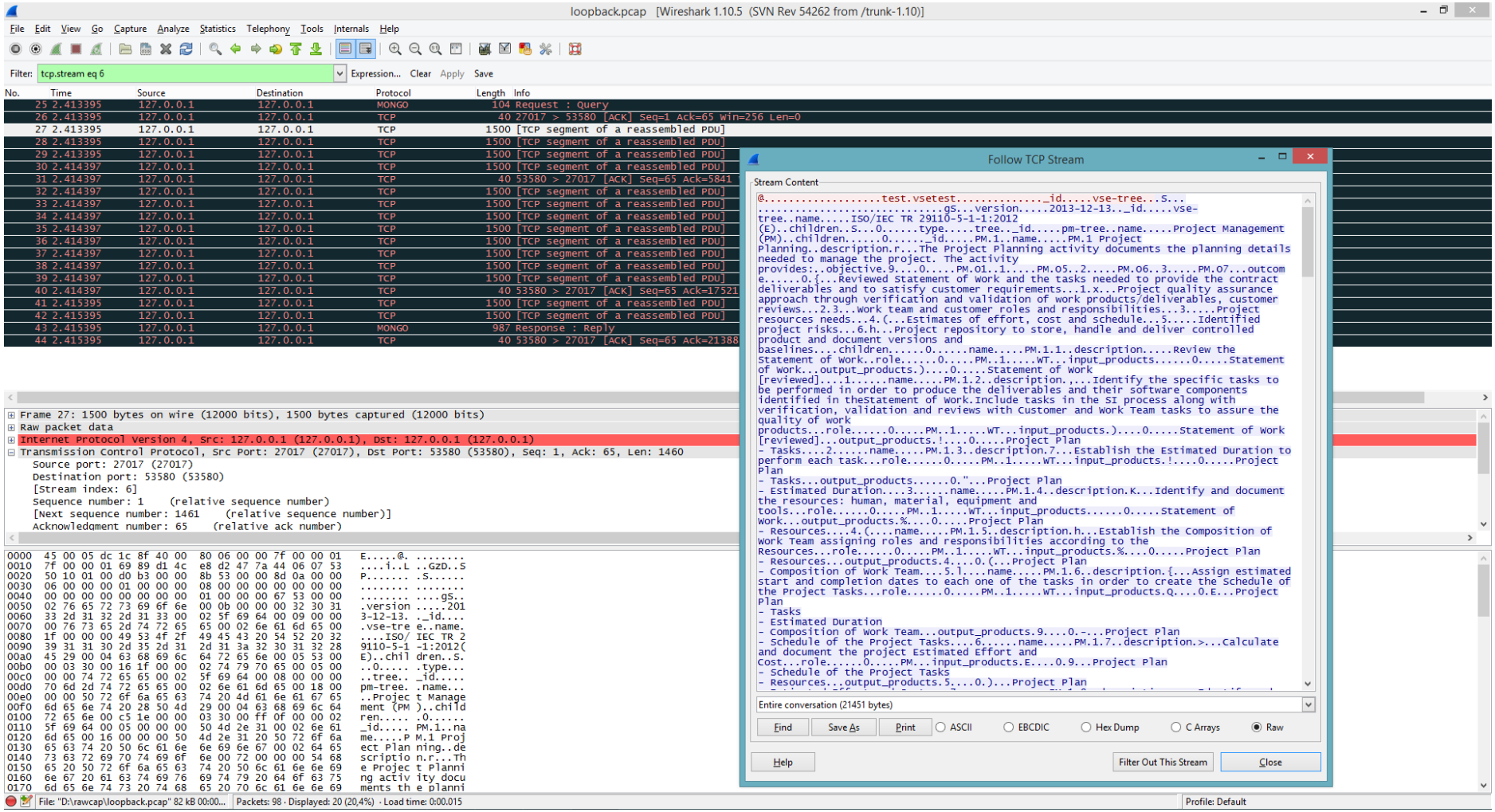
http://stackoverflow.com/questions/9728195/how-to-parse-json-data-when-the-property-name-is-not-known-in-advance

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#Compatiblity

Liite 5. MySQL Workbench -kaavio



Liite 6. RawCap-kaappaus



Liite 7. Työkalut

Dojo Toolkit - http://dojotoolkit.org/

Editorit:

- Netbeans IDE - https://netbeans.org/
- Notepad++ - http://notepad-plus-plus.org/

Google Chrome - http://www.google.com/chrome/
Google Chrome NetBeans Connector - https://chrome.google.com/webstore/detail/netbeans-connector/hafdlehgocfcodbgjnpecfajgkeejnaa

Java SE (JDK) - http://www.oracle.com/technetwork/java/javase/downloads/index.html

Kaavioiden piirtotyökalut:

- GenMyModel - http://genmymodel.com/
- Lucidchart - https://www.lucidchart.com/
- MySQL Workbench - http://www.mysql.com/products/workbench/
- WebSequenceDiagrams - https://www.websequencediagrams.com/

Microsoft Office 2007
Microsoft OneDrive - https://onedrive.live.com
Microsoft Windows 8/8.1 Pro x64 FI

Node.js - http://nodejs.org/

PDF-muunnostyökalut

- NitroCloud - https://www.gonitro.com/cloud/convert-pdf
- pdf2json - https://code.google.com/p/pdf2json/
- ThePDF - http://www.thepdf.com/convert-pdf-to-xml.html

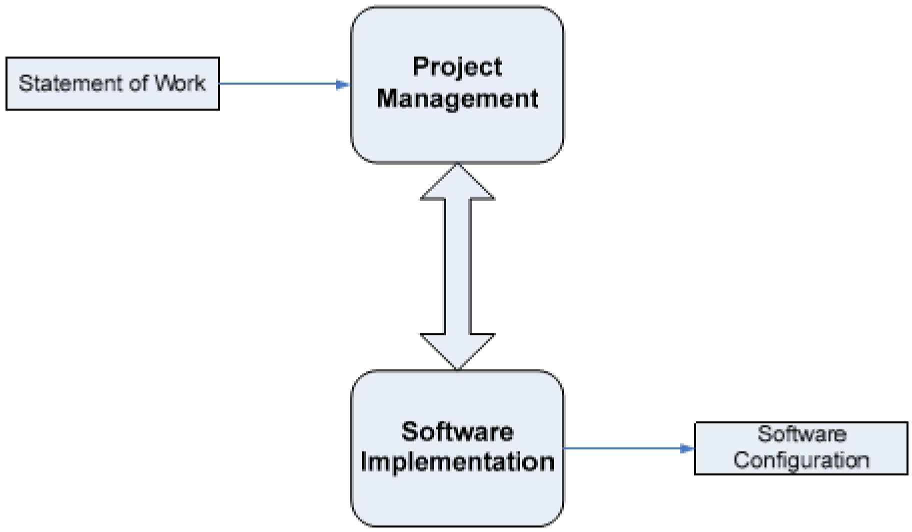
Tietokannat:

- MongoDB - http://www.mongodb.org/
- MySQL Community Server - http://www.mysql.com/

Tietoliikenteen kaappaus ja analysointi:

- RawCap - http://www.netresec.com/?page=RawCap
- WireShark - https://www.wireshark.org/

Liite 8. Koodattu kuva



Kuva 1 - PNG-muotoinen kuva

Listaus 1 - Kuva 1 base64 koodattuna XHTML-tunnisteiden sisällä


```
<img
src="
2AAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAA
C5NSURBVHhe7Z0HVBXX/rYTKzEq9i5oVLBjBQvFClasEQuKHU0sJNFgBRVFjYolCooKduw
dsBARG6JRUKJgBYyKJQm5SW5cd/3Xt/7fa/bc+fjGdpwZhOG8z3qX6zf77Jlz2OfMM3vmF
D/6X0IIIUaGHieEEGNDj5McJNZcSUlJkYaAkJyHHic6k5aW5u/vb2dn99FHzk6OZtnbGx
sLSwsPDwGhoeHS+NCSI5BjxPdyMzMnDzZx8raeorvzJi4hCe/vTDgodbWN
SIiIqQxIiQHoMeJPsTHxzdubBc-
QuEShM+ZyUmqffgM8PDxevHghDRYhukKPEx0ICwtzdHK5npquUBgjZ93GLfYOD1lZWdKQE
aIf9DjRSlRUdMfOrhmZWQpzMYrExCVA5ZyVE92hx4kmEhMTRO/1fZE4Szmtdm0bbd7r17
S2BGiE/Q40YSdnR3f0nyvdOvhfuDAAWn4CNEDepyoBz6ClRSeYt6ey0mpNra2vLpCdIQeJ
+qxd3CIPBGn8BTzzvTpN4CfRCQ6Qo8TlaSlpVlZWymysZWZ2iStRu2eHgMMlMaREM3Q40ly5c
v9x4/UWEoxpRkZGZZWFhI40iIZuhxopJp06bN9AtQCzJSGkhBt0ONEJV5ew
1euDlXoiTExjRrbJSYmPSkNJiDbocaISZxeX/YePK/TEmBhHJ+
....
(5 sivua myöhemmin)
....
LFi4iICC+v4ZaWpT4yY3B2snDhQv6ICvkA0OOEEGJs6HFCCDE29DghhBgbepwQQowNPU4I
IcaGHieEEGNDjxN-
CiLGhxwkhxNjQ44QQYmzocUIIMTb0OCGEGBt6nBBCjA09TgghxoYeJ4QQY0OPE0KIkfnf/
/2/TIQIeKvjh3IAAAAASUVORK5CYII="/>
```