Sisäänkirjautumisjärjestelmän ja käyttäjätietokannan luominen Node.js-ympäristössä

Samuli Vaittinen

2019 Laurea

Laurea-ammattikorkeakoulu

Sisäänkirjautumisjärjestelmän ja käyttäjä-tietokannan luominen Node.js-ympäristössä

Samuli Vaittinen
Tietojenkäsittely
Opinnäytetyö
Joulukuu, 2019

Laurea-ammattikorkeakoulu **Tiivistelmä**

Tietojenkäsittely (AMK)

Sisäänkirjautumisjärjestelmän ja käyttäjätietokannan luominen Node.js-ympäristössä

2019 Sivumäärä 22

Opinnäytetyön tavoitteena oli kehittää selainpeliin sisäänkirjautumis- ja tallennusjärjestelmä, joka tallentaa käyttäjän pelitiedot NoSQL-tietokantaan. Peli itsessään oli työn alkaessa jo valmis, mutta siitä puuttui tallennusominaisuus, joka mahdollistaisi pelin jatkamisen toisella koneella tai toisella selaimella. Tavoitteena oli tällä ominaisuuden lisäyksellä lisätä pelaajien pelaamiseen käyttämää aikaa ja siten hyödyttää pelaajien lisäksi toimeksiantajaa, jonka tavoitteena on jatkokehittää peliä kaupallisiin tarkoituksiin.

Työ on muodoltaan toiminnallinen ja sen suunnitteluvaihe sisältää myös tutkimuksellisia elementtejä. Sisäänkirjautumisen ja tallennuksen ratkaisuun löytyy useita erilaisia teknologioita ja mahdollisuuksia, joista valittiin taloudellisuudeltaan ja suoritustasoltaan sopivimmat vaihtoehdot. Työn tietoperustana on modernit web-teknologiat, joista tähän työhön on valittu Node.js, JavaScript, MongoDB ja Oauth2. Näitä menetelmällisiä ratkaisuja käsitellään sekä implementoinnin yhteydessä että yleisesti työn tietoperustassa ja sovelletaan toteutuksessa.

Lopputuloksena syntyi back end-ratkaisu, joka vastaa toimeksiantajan määrittelemiä tarpeita. Ratkaisu on myös tehty niin, että sitä on helppo ylläpitää ja, että siihen on mahdollista lisätä uusia lisäosia jatkokehityksen myötä. Tämän kaltainen back end-ratkaisu voidaan helposti skaalata suurelle joukolle käyttäjiä modernien pilvipalveluiden avulla.

Asiasanat: Web Development, Node.js, DevOps, OAuth2

Laurea University of Applied Sciences
Degree Programme in Business Information Technology
Bachelor's Thesis

Abstract

Authentication System and User Database Implementation in Node.js Runtime Environment

2019 Pages 22

The goal of this thesis was to develop a login and saving system for a web browser-based game, which would save the information into a NoSQL-database. The game itself was ready, when the work on this project started, but it lacked the saving functionality, which would allow the user to continue the game with another computer or web browser. By adding this feature, the goal was to extend the time that the players spend playing the game, which would benefit the players, but also the commissioner whose intention is to develop the game for commercial purposes.

The planning phase of the implementation consisted of elements of a research, but this study is a functional thesis. A vast amount of technologies and possibilities exists to solve this kind of a problem and the ones included in this project were chosen according to their economical features and processing power. The knowledge that the solution was based on includes the modern web technologies that were chosen for this projects implementation which were Node.js, JavaScript, MongoDB and Oauth2. These methodological solutions are covered in the fourth part of the thesis, but they are also covered generally in the research part before the implementation.

As a conclusion, a back end-solution was developed, which answers to the needs and expectations defined by the commissioner. The implementation is carried out and designed in a way that it is easy to manage and that adding new components is possible and convenient for the developers. A back end-solution of this type is scalable for a larger mass of users through a modern cloud platform.

Keywords: Web Development, Node.js, DevOps, OAuth2

Sisällys

1 Johdanto ... 6
2 Työn tavoite ... 6
3 Teoreettinen viitekehys ... 7
3.1 Tietojärjestelmien kehittäminen ... 7
3.2 Vesiputousmalli ja projektin jakaminen iteraatioihin ... 8
4 Kohdeympäristöt ... 8
4.1 Node.js-lisäosat ... 8
4.2 EJS ... 9
4.3 Express.js ... 9
4.4 Passport.js ... 9
4.5 MongoDB & Mongoose ... 10
4.6 OAuth2-autentikointi ... 10
5 Työn toteutus ... 12
5.1 Node.js-lisäosien asentaminen ... 12
5.2 Express-ohjelman luominen ja autentikointi-reitit ... 13
5.3 Node.js-reittien määrittäminen ... 14
5.4 Googlen OAuth-ominaisuuden liittäminen ... 16
5.5 Tietokannan lisääminen Node.js-ympäristöön ... 16
5.6 Tietokannan ja Passport.js-strategian yhdistäminen ... 18
5.7 Uniikin cookie:n luominen käyttäjälle ... 19
6 Yhteenveto ... 20
Lähteet ... 21
Kuviot ... 22

1 Johdanto

Vuonna 2019 IT-alan ammattilaisten keskuudessa suosittu ohjelmistokehitykseen keskittyvä keskustelufoorumi Stackoverflow julkaisi tutkimuksen, johon osallistui lähes 90 000 ohjelmistokehittäjää maailmanlaajuisesti. 20 minuuttia kestävässä kyselytutkimuksessa kartoitettiin muun muassa alan ammattilaisten valintoja käytettävien teknologioiden suhteen ja suosituimmaksi teknologiaksi valikoitui JavaScript -ohjelmointikieli, jonka valitsi 62.5 % vastanneista ensisijaiseksi teknologiaksi web-kehitykseen.

JavaScript, joka on alun perin kehitetty tuomaan dynaamista sisältöä verkkosivuille taipuu nykyään lisäosineen myös serveripuolen kattaviin full stack-ratkaisuihin. Sille kehitetään jatkuvasti lisäosia ja laajennuksia suurien IT-alan toimijoiden, kuten Googlen ja Facebookin toimesta ja perustellusti voidaan väittää, että JavaScript full stack-ratkaisuineen on web-kehityksen johtava trendi nyt ja lähitulevaisuudessa.

Opinnäytetyön alussa on kuvattu tämän implementoinnin taustalla oleva kirjallisuus ja teoreettinen viitekehys. Sen osalta olen pyrkinyt ottamaan huomioon modernin ohjelmistokehityksen yleiset käytänteet ja toimintatavat kehittäessäni tätä ohjelmiston osaa. Työ on jaettu vesiputousmallin mukaisesti iteraatioihin, joita käsitellään tarkemmin luvussa viisi.

Kohdeympäristössä käyn läpi tarkemmin, mistä komponenteista tämä ohjelma koostuu. Node.js-ympäristössä kehitystyötä tehdessä ei ole tarvetta kehittää uutta ratkaisua, koska useisiin ongelmiin löytyy ratkaisu valmiin lisäosan muodossa. Tässä työssä käytettiin lisäosia, jotka auttoivat muun muassa sisäänkirjautumislogiikan muodostamisessa, tietokantaan yhdistämisessä ja Node.js-valmiiden ominaisuuksien helpommassa käyttöönotossa.

Viidennessä kappaleessa käydään vaiheittain läpi, millä tavalla toteutuksen kokonaisuus on rakennettu. Teknisiä ratkaisuja on havainnollistettu kuvakaappauksin ohjelmakoodista ja toteutus käydään läpi iteraatioittain. Useat konseptit, kuten tietokannat, evästeet ja reititykset, joita tässä luvussa käsitellään ovat tuttuja web-kehityksen osalta jo vuosikymmenten takaa. Tavoitteena on, että tämä raportti antaa kuvan lukijalle, millä tavalla nämä tekniset ratkaisut voidaan implementoida modernissa kehitysympäristössä viimeisintä teknologiaa hyödyntäen.

2 Työn tavoite

Opinnäytetyön toimeksiantaja on suomalainen yritys, joka työllistää kaksi omistajaansa, on ohjelmistokehitykseen ja sisällöntuottamiseen keskittynyt yritys. Yrityksen kehittämä selaimessa toimiva JavaScriptillä ohjelmoitu peli tallentaa pelin tiedot selaimen eväisteisiin. Käytännössä tämä tarkoittaa sitä, että peliä ei voi jatkaa samasta kohtaa toisella selaimella tai tietokoneella.

Sain freelance-työnä ratkaistavaksi tämän ongelman ja päädyin ratkaisuna luomaan sisäänkirjautumis- ja käyttäjätietokantajärjestelmän, joka hyödyntää sosiaalisen median ohjelmointi rajapintoja ja OAuth2-autorisointia. Ratkaisu toimii Node.js-ympäristössä, joka mahdollistaa JavaScriptin suorittamisen palvelimella. Käyttäjän ei tarvitse erikseen luoda profiilia peliä varten, vaan riittää, että henkilöllä on joko Facebook- tai Google+ -tunnukset. Peli- ja käyttäjätiedot tallennetaan NoSQL-tietokantaan, JSON-objektina (JavaScript Object Notation), joka on yksinkertainen ja yleisesti käytetty tiedostomuoto web-ohjelmien väliseen tiedonvälitykseen. Lopputuloksena syntyi edellä mainittuja teknologioita hyödyntävä web-ohjelman osa, jonka avulla pelin tallennus onnistuu liittämällä tallennustiedot sosiaalisen median profiilin tietoihin tavoitteiden mukaisesti.

3 Teoreettinen viitekehys

Teoreettisena viitekehyksenä tässä työssä on tietojärjestelmien ja ohjelmistotuotannon kehittämistä käsittelevä kirjallisuus (Tietojärjestelmien kehittäminen, Ohjelmistotuotannon käytännöt, Ohjelmistoprojektin sudenkuopat ja miten ne vältetään) ja sen päämetodiin, eli suunnittelututkimukseen (Design Science Research) liittyvä kirjallisuus. Ohjelmistokehityksen käytäntöihin, suunnitteluun ja riskienhallintaan liittyvään kirjallisuuteen sisältyy laaja-alaisesti erilaisia lähestymiskulmia, joiden käyttötarkoitus vaihtelee projektien koon ja organisaation rakenteen mukaan. Tähän opinnäytetyöhön käytettävien teorioiden valinnassa kriteerinä on ollut, että ne voidaan yhdistää suunnittelututkimuksen implementoinnin kanssa kokonaisuudeksi ja, että ne soveltuvat ketterään ja dynaamiseen ohjelmistokehitykseen.

3.1 Tietojärjestelmien kehittäminen

Pohjonen (2002, 14) määrittelee tietojärjestelmien kehittämisen osaksi sitä suorittavan organisaation toiminnan kehittämiseksi, jolle voidaan asettaa tavoitteeksi:

- Se auttaa toimintayksikköä suuntautumaan tavoitteisiinsa entistä paremmin
- Se mahdollistaa entistä vaativampien tavoitteiden asettamisen
- Se tekee mahdolliseksi jonkin uuden toiminnon
- Se tehostaa jo olemassa olevia toimintatapoja

Toiminnon kehitys kohdistuu joko ihmisiin, teknologiaan tai toimintoihin. Vaikka tietojenkäsittelyn kehittäminen toimintona vaikuttaa yleensä kaikkiin näihin osa-alueisiin, on tässä opinnäytetyössä keskiössä uuden toiminnan, eli käyttäjätietokannan ja tallennusmahdollisuuden lisääminen, sekä entistä vaativampien tavoitteiden asettamisen mahdollistaminen. Näitä tavoitteita ovat yritys X:n suunnitelmat kasvattaa tulevaisuudessa mainostulojaan kohdennetulla mainonnalla, sekä uusien ja maksullisten palveluiden luominen rekisteröityneille käyttäjille.

Kuten Pohjonen (2002) osuvasti tuo esille, niin asiakasvaatimusten kerääminen on vaativaa työtä. Myös tässä projektissa vaatimusmäärittelyssä otettiin huomioon toiminnalliset vaatimukset, eli miten järjestelmä toimii ulkoapäin tarkasteltuna, miten se kommunikoi ympäristönsä kanssa ja miten ne työskentelevät sen kanssa (Pohjonen 2002, 28). Tämän lisäksi, koska kyseessä oli olemassa olevan digitaalisen palvelun jatkokehittäminen, huomioon otettiin myös ei-toiminnallisia vaatimuksia, kuten millä tavalla pelin tiedot tallentuvat tällä hetkellä (JSON) ja millä teknologiaa ja ohjelmointikieltä pelin kehittämiseen ja jakamiseen on käytetty (JavaScript ja Nginx-palvelinohjelma hostattuna yrityksen X palvelimelle). Näiden speksien muodostaminen ei jälkeenpäin ajateltuna ollut niin suoraviivaista kuin kirjoitetusta tekstistä voi käydä ilmi. Toimivan kokonaisuuden rakentamista varten ja näiden ehtojen selvittämiseksi käytiin useita tapaamisia yrityksen X edustajien kanssa.

3.2 Vesiputousmalli ja projektin jakaminen iteraatioihin

Opinnäytetyöprojektin käytännön toteutus on jaettu iteraatioihin vesiputousmallin modernimman tulkinnan mukaisesti, joka sallii edellisiin työvaiheisiin palaamisen, eli taaksepäin iteroinnin (Haikala ym. 2011, 37). Ohjelmiston ohjelmointiosuus on jaettu seuraaviin iteraatioihin: Node.js-lisäosien asentaminen, Express-ohjelman luominen ja autentikointi reitit, Node.js-reittien määrittäminen, Googlen OAuth-ominaisuuden liittäminen, tietokannan lisääminen Node.js-ympäristöön, tietokannan ja Passport.js-strategian yhdistäminen, uniikin evästeen luominen käyttäjälle.

Vaikka tässä projektissa toteutetaan samalla kevyt käyttöliittymä ohjelmiston testaamista varten, niin tullaan tämä käyttöliittymä korvaamaan jo kehitetyllä pelivalikolla ohjelmiston integroinnin aikana.

4 Kohdeympäristöt

Node.js on yksi käytetyimmistä moderneista web-teknologioista, joka mahdollistaa JavaScript-koodin suorittamisen palvelimella. Sen avoimeen lähdekoodiin perustuva ratkaisu perustuu Googlen kehittämään V8-moottoriin, joka kääntää JavaScript-koodia konekieleksi. Kyseessä ei ole sinänsä Apache:n tai Nginx:iin verrattavasta palvelinohjelmasta, vaan palvelimella käytettävästä runtime-ympäristöstä. Laajennuksien avulla Node.js-ympäristöä voidaan kuitenkin käyttää palvelinohjelman tavoin.

4.1 Node.js-lisäosat

Node.js:lle on saatavilla useita eri laajennuksia, jotka voidaan asentaa helposti komentorivillä komennolla npm install paketti, jossa sana paketti korvataan paketin nimellä. Käyttäjän on tällöin myös oltava Node.js-projektin juurihakemistossa. Node Package Manager, eli npm tulee valmiiksi asennettuna Node.js-ympäristön kanssa.

Opinnäytetyöprojektissa käytettävät laajennukset esimerkiksi tekevät API-kutsuja, käsittelevät tietokantaliikennettä tai tuovat Node.js-ympäristössä abstrahoituja ominaisuuksia käyttäjän työkalupakettiin. Alla on lyhyesti esitelty jokainen tässä projektissa käytetty laajennus. Yksityiskohtaisemmin näiden laajennusten käyttö ja tarkoitus tässä projektissa tulee ilmi myöhemmissä kappaleissa, joissa käsitellään opinnäytetyöprojektin iteraatioita ja käytännön toteutusta.

4.2 EJS

EJS (Embedded JavaScript Templates) on Node.js-ympäristössä käytettävä template engine -lisäosa, joka mahdollistaa staattisten html-mallien käytön Node.js-ympäristössä. Käytännössä EJS on ohjelmiston osa, joka liittää datan valmiisiin html-sivuihin muuttujien avulla, jolloin sivujen suunnittelu ja valmiiden mallien käyttäminen on vaivattomampaa.

4.3 Express.js

Express.js on Node.js-ympäristössä yleisesti käytetty ohjelmistokehys-lisäosa, joka mahdollistaa lukuisilla työkaluillaan helpomman käytettävyyden ja Node.js-ympäristön käytön web-serverinä. Express.js tunnetuimmat ominaisuudet ovat muun muassa reititys, jolla määritetään miten Node.js-ohjelma vastaa clientin lähettämään pyyntöön, sekä middlewaren käyttäminen. Käsitteellinen ero Node.js:n ja Express.js:n välillä on abstrahoinnin taso. Node.js on serverillä toimiva JavaScriptiä suorittava ympäristö ja Express.js on Node.js:n toimintoja hyväksikäyttävä ohjelmistokehys, joka tekee Node.js:n ominaisuuksien käyttöönotosta helpompaa

Middleware-funktioilla on pääsy request object ja response object tietoihin, joita voidaan muokata middelware-koodilla. Näihin objekteihin voi lisätä selaimesta serverille lähetettävää dataa (request object) tai vaihtoehtoisesti serveriltä selaimeen (response object) lähetettävää dataa. Käytännössä tämä on yleinen tapa toteuttaa esimerkiksi tietokantaliikenteeseen, autentikointiin ja muuhun toiminnallisuuteen liittyvä ohjelmakoodi Node.js-ympäristössä.

4.4 Passport.js

Tässä opinnäytetyössä korostuu erityisesti käyttäjien autentikoinnin ja käyttäjätietojen tietokantaan lisäämisen implementointi, jossa Passport.js-laajennus on hyvin merkittävässä osassa. Passport.js on itsessään middleware-funktio, joka vaatii toimiakseen Express.js laajennuksen.

Passport.js tarjoaa yli 500 erilaista "strategiaa", joilla voidaan toteuttaa käyttäjän autentikointi, joko sosiaalisen median profiilin tai perinteisemmin käyttäjänimi ja salasana -kombinaation avulla. Käytettäessä esimerkiksi sosiaalisen median profiilin tietoja sisäänkirjautumisessa, Passport.js hoitaa automaattisesti keskustelun näiden kolmannen osapuolten rajapintojen välityksellä, mikä tarkoittaa ohjelmistokehittäjälle pienempää työmäärää. Tämän lisäksi

Passport.js sisältää monia muita pieniä apufunktioita ja ohjelmanosia, jotka esimerkiksi pitävät käyttäjän kirjautuneena session aikana, sekä palvelusta uloskirjautumisen ja joilla voidaan helposti toteuttaa Oauth2-autentikointiprotokollan mukainen sisäänkirjautuminen, johon paneudutaan tarkemmin seuraavissa luvuissa.

4.5 MongoDB & Mongoose

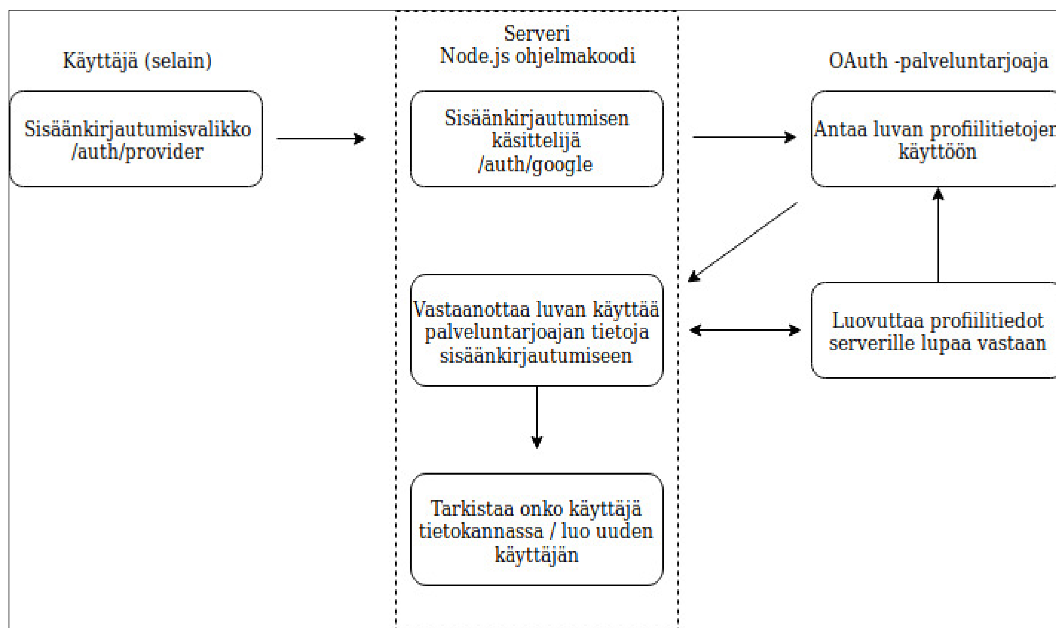
MongoDB on yleisesti käytetty NoSQL-tietokantaohjelma, joka tallentaa datan JSON (JavaScript Object Notation)-objektina. Tämä tekee sen käytöstä erityisen ketterää JavaScript-kehityksessä, jossa JSON on yleinen tapa esittää dataa (esimerkiksi tässä opinnäytetyössä käytetään JSON-muotoa usealla eri tavalla). JSON on tästä huolimatta JavaScriptistä riippumaton tiedostomuoto. MongoDB luo automaattisesti jokaisen käyttäjän erottavan uniqueID-tunnisteen. Mongoose on Node.js-laajennus, joka mahdollistaa MongoDB-objektien helpon mallintamisen, sekä tietokantaan yhdistämisen ja tietokantaliikenteen hallinnan.

4.6 OAuth2-autentikointi

Vuonna 2006 Twitterillä työskennellyt Blaine Cook oli yhteydessä työkavereihinsa tarkoituksenaan toteuttaa sisäänkirjautuminen Twitterin ohjelmistorajapinnan kautta. He huomasivat pian, että sen aikaiset autentikointiprotokollat eivät tukeneet tämän kaltaista ratkaisua ja he jatkoivat keskustelua aiheesta ja olivat yhteydessä myös Googlen kehittäjiin. Vuoden 2007 loppuun mennessä ensimmäinen versio nimeltään "OAuth Core 1.0" oli valmis. OAuth 1.0 ja OAuth 2.0-protokollat ovat määritelty dokumenteissa RFC 5849 ja RFC 6749.

Tätä opinnäytetyötä kirjoittaessa, vuonna 2019, OAuth-protokollaan perustuvat ratkaisut ovat arkipäivää monelle web-palveluiden käyttäjille ja useat sivut tarjoavat mahdollisuuden kirjautua sosiaalisen median tunnuksilla palveluihinsa. Verrattuna perinteiseen sisäänkirjautumiseen, jossa käyttäjä valitsee käyttäjänimen ja salasanan palvelukohtaisesti, OAuth-protokolla tarjoaa mahdollisuuden tunnusten luomiseen ja sisäänkirjautumiseen, ilman että käyttäjän tarvitsee paljastaa salasanaansa tai muita tietoja itsestään (Eran Hammer-Lahav, 2007).

Oheisessa kaaviossa on esitetty OAuth-sisäänkirjautumisprosessi tämän opinnäytetyön näkökulmasta.



Kuvio 1: OAuth flow

Keskellä rajattuna ovat prosessit, jotka tapahtuvat palvelimella ja Node.js-ympäristössä. Oikealla olevalla OAuth-palveluntarjoajalla viitataan OAuth-sisäänkirjautumista tarjoavaan, esimerkiksi sosiaalisen median palveluun, kuten Twitter, Google tai Facebook. Vasemmalla on käyttäjän tekemä sisäänkirjautumispyyntö.

Erityistä huomiota ohjelmistokehittämisen kannalta on syytä kiinnittää vaiheeseen, jossa palveluntarjoajaan ollaan yhteydessä palvelimelta. Kun käyttäjä kirjautuu sosiaalisen median tunnuksilla sisään, ottaa Node.js-ohjelma yhteyttä OAuth-palveluntarjoajaan saadakseen luvan tähän tarkoitetun koodin muodossa. Tämä koodi lähetetään Node.js-ohjelmalle, joka käyttää tätä sen jälkeen profiilitietojen hakemiseen palveluntarjoajalta. Tämän jälkeen palveluntarjoajan tiedot ovat käytettävissä JSON-tiedostomuotona Node.js-ympäristössä, jolloin näiden tietojen perusteella voidaan tehdä valittuja toimenpiteitä käyttäjätietokannassa. Tarkka lukija saattaa miettiä miksi palveluntarjoaja ei lähetä profiilin tietoja ensimmäisessä vastauksessa käyttäjän autentikointipyyntöön. Syy tähän on OAuth-protokollan käytännöissä, jossa resursseja ja autentikointia hallitaan kahden eri serverin kautta. Resursseihin pääseminen vaatii siis erillisen autentikointiserverin myöntämän access token-avaimen ja myös joissain tilanteissa lisäksi refresh token-avaimen, jolla voidaan pidentää istunnon pituutta. Turvallisuuteen liittyvistä syistä refresh tokenin käyttöä on syytä välttää aina, kun autentikoinnilla on tarkoitus suojata salattavaksi tarkoitettua tietoa (Ratros Y, 2018).

Tässä opinnäytetyössä onnistuneen autentikoinnin jälkeen tarkistetaan, onko käyttäjä kirjautunut aikaisemmin, ja mikäli ei ole, niin luodaan uusi käyttäjä tietokantaan palveluntarjoajan antamien tietojen perusteella. Selainpeli, johon tämä sisäänkirjautumisjärjestelmä luodaan tallentaa jo valmiiksi pelitilan JSON-objektina, jolloin se on helppo liittää vastaanotettuihin käyttäjätietoihin, jotka ovat samassa tiedostomuodossa.

5 Työn toteutus

Työn toteuttaminen on jaettu kuuteen eri työvaiheeseen, eli iteraatioon. Nämä työvaiheet käydään tässä kappaleessa siinä järjestyksessä läpi, jossa ne implementoidaan. Ensimmäisessä vaiheessa määrittelemme Node.js-ympäristön sisäiset komponentit, sekä yleisen viitekehyksen toteuttamista varten.

5.1 Node.js-lisäosien asentaminen

Työ toteutettiin käyttäen Atom tekstieditoria Linux-ympäristössä (Ubuntu). Ensimmäisenä kirjoitetaan komentoriviin komento "npm init", joka luo uuden projektin ja kehitysympäristön juurihakemistoon. Edellisen komennon käyttäminen edellyttää, että käyttäjältä löytyy Node Package Manager ja muut vaadittavat liitännäiset valmiiksi asennettuna. Muiden tarvittavien lisäosien installointi onnistui samalla logiikalla projektin juurihakemistossa, jonka jälkeen Node.js-tiedosto package.json näytti tältä:

```
{
  "name": "oauthlogin",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cookie-session": "^1.3.3",
    "ejs": "^2.7.1",
    "express": "^4.17.1",
    "mongoose": "^5.7.9",
    "nodemon": "^1.19.4",
    "passport": "^0.4.0",
    "passport-google-oauth20": "^2.0.0"
  }
}
```

Kuvio 2: Package.json-tiedosto ja Node.js-liitännäiset

Package.json-tiedosto on erityisen hyödyllinen liitännäisten- sekä versionhallinnan kannalta. Tämä tiedosto määrittää, mitkä liitännäiset asennetaan automaattisesti esimerkiksi silloin kun ohjelma ladataan pilvipalveluun suoritettavaksi. Siitä näkee myös millä Node.js-versiolla se on implementoitu.

Nämä lisäosat kutsutaan Node.js-ympäristöön syntaksilla "const *laajennuksen nimi* = require('*laajennuksen nimi*');". Hyvien käytäntöjen mukaan tämä syntaksi lisätään heti ohjelmakoodin alkuun yläriville. Kun tähän työhön tarvittavat laajennukset ovat kutsuttu, niin näyttää ylärivi tältä:

```
const express = require('express');
const authRoutes = require('./routes/auth-routes');
const profileRoutes = require('./routes/profile-routes');
const passportSetup = require('./config/passport-setup');
const mongoose = require('mongoose');
const keys = require('./config/keys');
const cookieSession = require('cookie-session');
const passport = require('passport');
```

Kuvio 3: Lisäosien lisääminen toteutukseen

Tässä kuvassa näkyy myös, että ympäristöön on kutsuttu esimerkiksi tiedostoja poluista "./config/keys" ja "./config/passport-setup". Näihin tiedostopolkuihin lisätään myöhemmissä iteraatiovaiheissa ohjelmaa suorittavia funktioita ja API-tunnistukseen tarvittavia parametrejä.

5.2 Express-ohjelman luominen ja autentikointi-reitit

Express laajennuksen avulla saadaan useita tämän ohjelman toteutuksen kannalta olennaisia lisäosia käyttöön. Laajennus asennettiin edellisen iteraation aikana ja se otetaan käyttöön seuraavalla tavalla:

```
// Express
const app = express();

app.set('view engine', 'ejs');
```

Kuvio 4: Express.js

Kuviossa 4 näkyvä funktio luo Express.js-applikaation Node.js-ympäristöön, jonka jälkeen voimme luoda esimerkiksi näkymämoottorin (view engine), joka on yksi Express.js:n ominaisuuksista, joita tässä ohjelmistossa tarvitaan. Tämän jälkeen ohjelmalle määritellään kuunneltava portti, jossa Node.js-ympäristö suoritetaan testausta varten, sekä lisätään juurihakemisto ja näkymä, joka renderöidään tässä määritellyssä portissa:

```
// Juurihakemisto
app.get('/', (req, res) => {
  res.render('home');
});


// Ohjelman käynnistys
app.listen(3000, () => {
  console.log('Now listening port 3000');
});
```

Kuvio 5: Juurihakemiston ja portin määrittäminen

5.3 Node.js-reittien määrittäminen

Tässä iteraatiovaiheessa ohjelmaan määritellään reitit, joiden avulla Node.js-ympäristöön lisätään sisäistä toiminnallisuutta. Ensimmäiseksi kutsumme Express.js:n sisään rakennettua Router()-funktiota ja tallennamme sen muuttujaan router:

```
const router = require('express').Router();
```

Kuvio 6: Reuter()-funktio

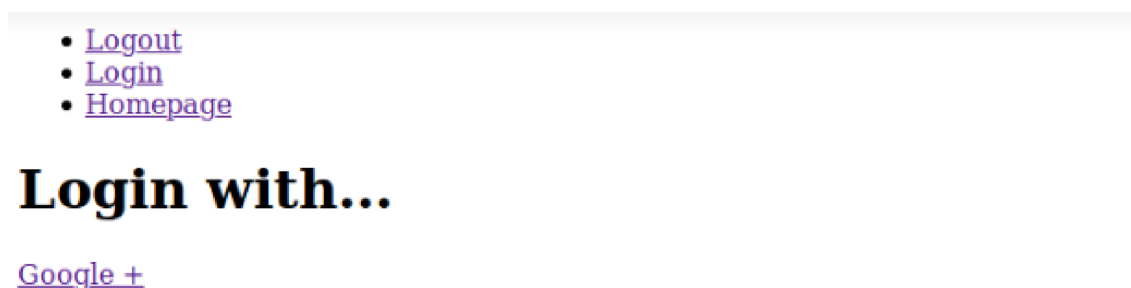
Jokainen reitti ottaa request- (req) ja response- (res) objektit parametreiksi. Request-objekti sisältää kaiken sen tiedon mitä käyttäjä lähettää verkkoselaimellaan serverin käsiteltäväksi. Esimerkiksi ulos kirjautuessa request-objektiin liitetään logout()-funktio, jonka avulla palvelimelle tulee käsky suorittaa kyseinen funktio. Response-objektiin liitetään taas tiedot palvelimelta asiakkaalle lähtevistä tiedoista. Kuvassa 7 res-objektiin on lisätty esimerkiksi käsky ohjata käyttäjä toiselle sivulle ja renderöidä juurihakemistossa oleva html-sivu.

Oheisessa kuvassa on jaoteltu reitit ja lisätty niiden yläpuolelle kommentit kertomaan, mitä kyseisessä reitissä tapahtuu ohjelman suorituksen aikana:

```
// sisäänkirjautuminen

router.get('/login', (req,res) => {
  res.render('login');
});

// uloskirjautuminen

router.get('/logout', (req,res) => {
  req.logout();
  res.redirect('/');
});

// autorisointi googlen API:n kautta

router.get('/google', passport.authenticate('google', {
  scope: ['profile']
}));

//  callback reitti googlen autorisoinnille

router.get('/google/redirect', passport.authenticate('google'), (req, res) => {
    res.send(req.user+'  Tämä sivu on callback URI');
});
```

Kuvio 7: Node.js-reitit

Ensimmäinen reitti on kirjautumissivu, joka renderöi yksinkertaisen html-valikon testausta varten:



Kuvio 8: Html-käyttöliittymä testausta varten

Tällä sivulla voidaan testata kirjautumislogiikan toimivuutta ja muiden reittien renderöitymistä. Tämä käyttöliittymä tullaan kuitenkin korvaamaan integraatiovaiheessa lopputuotteessa olevalla käyttöliittymällä, joka sisältää kehittyneempää grafiikkaa ja toiminallisuutta.

Logout-reitti sisältää uloskirjautumisen, jonka toiminnallisuus on itseään hyvin kuvaava. Reitti suorittaa logout-funktion, jonka jälkeen se palauttaa näkymän sisäänkirjautumissivulle. Seuraavissa reiteissä, joissa käytetään Passport.js-lisäosaa Node.js auktorisoi käyttäjän Googlen API:n kautta. Kohdassa "scope" on määritelty mitä parametrejä profiilista haetaan. Tässä kontekstissa profiilista haetaan profiilin perustiedot (nimi, ikä, sähköpostiosoite, profiilikuva),

jolla voidaan myöhemmin tunnistaa käyttäjä ja liittää hänen tietoihinsa tallennettuja tietoja tietokantaan. Nämä tiedot saadaan käyttöön, kun tämän reitin kautta vastaanotetulla koodilla haetaan tiedot seuraavassa callback-reitissä. Tämä logiikka vastaa kuvio1 esitettyä kohtaa, jossa OAuth-palveluntarjoaja antaa luvan profiilitietojen käyttöön.

5.4 Googlen OAuth-ominaisuuden liittäminen

Voimme käyttää Googlen tarjoamaa OAuth-sisäänkirjautumista Passport.js-komponentin avulla. Kun API-tiedot ovat konfiguroitu Googlen ohjelmistokehittäjille luodussa console.developers.google.com-palvelussa, niin saadaan käyttöön clientID- ja clientSecret-avaimet, jotka lisätään niille luotuun keys.js-kansioon JSON-objektina:

```
google: {
  clientID: '323560237764-ei65f9j1bi96er13upnv59eb47qck6pu.apps.googleusercontent.com',
  clientSecret: '5nlW4XVURIJcY4N5auxbVWoA'
},
```

Kuvio 9: Keys.js

Kuvion 9 clientID- ja clientSecret-avaimet ovat vaihdettu viimeisessä ohjelmiston versiossa ja yllä olevat ovat vain toimimattomia esimerkkejä havainnollistamaan Googlen tarjoaman API-palvelun toimivuutta.

5.5 Tietokannan lisääminen Node.js-ympäristöön

Tässä työssä käytämme MongoDB-tietokantaa, johon voimme tallentaa JSON-muodossa olevia dataobjekteja. Tallennukseen ja tietokantakyselyihin käytetään mongoose-lisäosaa, joka sisältää laajan valikoiman apufunktioita MongoDB-tietokannan hallitsemiseen. Ensimmäiseksi luomme schema-nimisen objektin, johon määrittelemme tallennettavan objektin muuttujat ja niiden datatyypit. Schema on ikään kuin pohjapiirros sille, miltä käyttäjien (tai tallennettavan objektin) pitää näyttää tallennettaessa datan näkökulmasta. Tähän työhön tarvitaan ainoastaan muuttujat käyttäjänimi ja googleId string (eli merkkijono) - muodossa:

```
const mongoose = require('mongoose');
const Schema  = mongoose.Schema;

const userSchema = new Schema({
  username: String,
  googleId: String
});

const User = mongoose.model('user', userSchema);

module.exports = User;
```

Kuvio 10: MongoDB:n implementointi

Laajempi ohjelmakokonaisuus, johon tämä työ liitetään osaksi tallentaa pelin tiedot myös JSON-objektina. Integrointi vaiheessa tähän lisätään pelin tilaan liittyviä muuttujia, jotka nyt tallentuvat käyttäjän evästeisiin.

MongoDB-tietokanta voidaan luoda paikallisesti, jolloin se voidaan yhdistää Node.js-tyyppiseen kehitysympäristöön paikallisesti toimivan web-palvelun ja portin kautta (esim. localhost:3000). Hyvin tavallista on myös, että ladattaessa koko valmis ohjelma pilvipalveluun (esim. Heroku Amazon Web Services, Azure) tallennetaan tietokanta tähän samaan pilveen osana kokonaispalvelua. Tässä työssä keskitytään laajemman kokonaisuuden pienempään osaan, joten työn lataaminen pilvipalveluun olisi turhan raskasta. Ratkaisuna tähän käytän ilmaista MongoDB Atlas web-palvelua, johon voi tallentaa MongoDB-tietokannan ilmaiseksi. Tähän tarvitsee vain Atlaksesta konfiguroinnin yhteydessä saatavat kirjautumistunnukset, sekä linkin, jota kautta ohjelma on yhteydessä tietokantaan:

```
mongodb: {
  dbURI: 'mongodb+srv://admin:admin@cluster0-gvlxd.mongodb.net/test?retryWrites=true&w=majority'
},
```

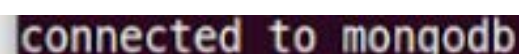
Kuvio 11: MongoDB-kirjautumistiedot

Testiä varten kirjautumistiedoiksi on valittu käyttäjänimeksi "admin" ja salasanaksi "admin". Hyvien tietoturvakäytänteiden mukaisesti nämä ovat vaihdettu lopullisessa versiossa. Hyvin todennäköistä kuitenkin on, että kun ohjelma kokonaisuutena on valmis, niin tullaan myös tietokanta tallentamaan toiseen palveluun.

```
// Mongodb
mongoose.connect(keys.mongodb.dbURI, () => {
  console.log('connected to mongodb');
});
```

Kuvio 12: MongoDB-yhdistäminen

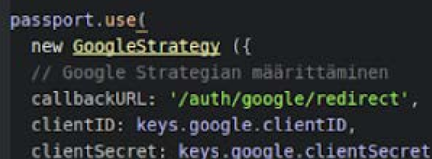
Tietokantaan yhdistäminen onnistuu seuraavalla call back-funktiolla, jonka toteuttaminen on hyvin yksinkertaista mongoose-lisäosalla. Kun mongoose on yhdistänyt ohjelman tietokantaan käyttäen kirjautumistietoja, niin tästä seurauksena laukeaa call back-funktio, joka kirjoittaa lokitietoihin viestin "connected to mongodb". Tästä tiedämme, että ohjelma toimii oikein ja saamme yhteyden tietokantaan.



Kuvio 13: 'connected to mongodb'

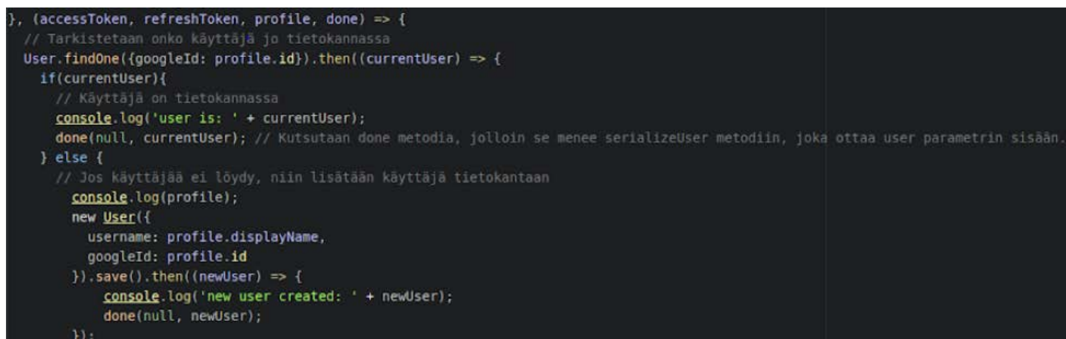
5.6 Tietokannan ja Passport.js-strategian yhdistäminen

Passport.js-lisäosa sisältää yli tuhat erilaista sisäänkirjautumisstrategiaa, jotka helpottavat sisäänkirjautumis- ja autentikointilogiikan hallitsemista ohjelmistoja kehittäessä. Sen käyttöönotossa määritellään callbackURL, johon käyttäjä uudelleenohjataan sisäänkirjautumisen onnistuessa, sekä clientID- ja clientSecret-avaimet, joita tarvitaan käyttäjätietojen hakemiseen Googlen API:n kautta. Nämä parametrit ovat määriteltynä funktion sisään, johon lisätään myös käyttäjätietojen tallentaminen tietokantaan:



Kuvio 14: Google-strategia

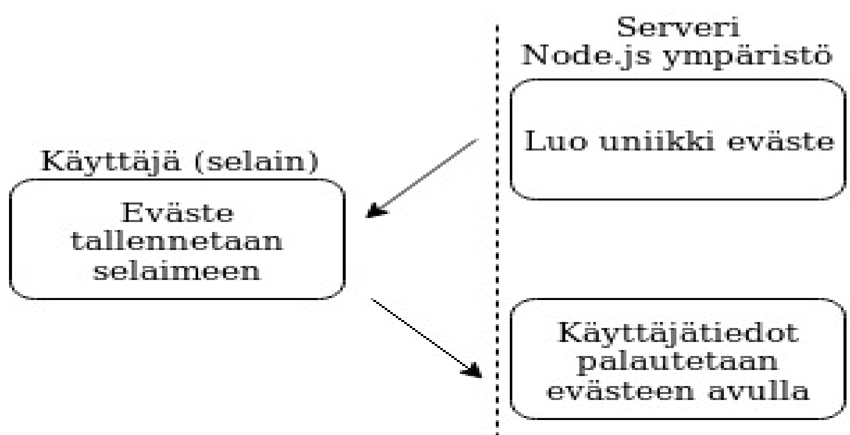
Kun tähän lisätään osaksi call back-funktio, niin voimme määritellä tehtävät toimenpiteet, jotka suoritetaan aina tämän funktion käytön yhteydessä. Käytännössä tämä tarkoittaa implementoituna, että kun henkilö kirjautuu sisään käyttäen Google-tunnuksiaan, niin tarkistetaan, onko kyseinen henkilö tietokannassa. Jos henkilön tiedot ovat tietokannassa, niin voimme palauttaa häneen liitettyä dataa tietokannasta selaimen käyttöön (esimerkiksi profiilikuvan tai keskustelufoorumin viestit). Jos henkilö ei ole kirjautunut aikaisemmin, niin luomme hänelle uuden tietueen tietokantaan.



Kuvio 15: Tietokantakyselyt call back-funktion sisällä

5.7 Uniikin cookie:n luominen käyttäjälle

Viimeinen lisäominaisuus, jonka teemme tähän työhön on sisäänkirjautumisstatuksen ylläpitäminen evästeiden avulla. Tällä tavalla käyttäjä voi jatkaa sovelluksen käyttöä niin kauan, kun eväste on voimassa. Evästeen luomiseen tarvitaan käyttäjän yksilöivä tieto. Yksi vaihtoehto olisi käyttää Facebook-tunnuksessa olevaa uniqueID-tunnistetta. Ongelmana tässä on se, että mikäli työtä jatkokehitetään niin, että siihen liitetään useamman sosiaalisen median kirjautumisvaihtoehdot, niin tuottaa tämän ominaisuuden liittäminen ylimääräistä vaivaa ja koodia ohjelmaan. Sen sijaan tässä työssä käytetään MongoDB:n luomaa id:tä, joka on jokaiselle käyttäjälle uniikki riippumatta siitä, minkä palveluntarjoajan kautta autentikointi on tehty.



Kuva 1: Evästeiden vaihtaminen

Passport.js tarjoaa meille valmiiksi sisäänrakennetut serializeUser- ja deserializeUser-funktiot, joilla liitetään evästeisiin tunnistetietoja.

```
passport.serializeUser((user, done)=> {
  done(null, user.id); // SerializeUser ottaa user.id ja serialisoi sen cookiehin.
  // tämä tapahtuu done metodin kautta, jonka serializeUser ottaa parametriksi.
});

passport.deserializeUser((id, done)=> {
  User.findById(id).then((user) => {
    done(null, user.id);
  });
   // Kun cookie tulee takaisin selaimella niin deserialisoi se ja etsi se id:llä tietokannasta.
});
```

Kuva 2: Käyttäjätietojen tallentaminen evästeisiin

Ensimmäisessä funktiossa liitetään user.id tunnisteen evästeeseen, kun käyttäjä on löytynyt tai tallennettu tietokantaan. Toisessa funktiossa tietokannasta etistään käyttäjä tunnistetiedon perusteella ja palautetaan käyttäjän tiedot tunnistetieto löydettäessä. Tämän lisäksi konfiguroidaan vielä ohjelmaan sisälle evästetiedot seuraavalla funktiolla:

```
app.use(cookieSession({
  maxAge: 24*60*60*1000,
  keys: [keys.session.cookieKey]
}));
```

Kuva 3: Evästetietojen konfigurointi

Tämän jälkeen käyttäjä pysyy sisällä kirjautuneena evästeiden avulla maxAge-muuttujaan tallennetun ajan verran, ellei hän kirjaudu ulos painamalla logout-linkkiä käyttöliittymässä.

6 Yhteenveto

Tämän työn tavoitteena oli luoda Node.js-ympäristössä toimiva sovellus, joka suorittaa sisäänkirjautumisen käyttäen sosiaalisen median palvelun OAuth2-autentikointia. Implementointi on toteutettu Passport.js-lisäosalla niin, että useampien kirjautumispalveluiden lisääminen on helppoa. Tämän lisäksi ympäristöön on mahdollista liittää JSON-muodossa dataa tallentava pelikokonaisuus, joka voidaan suorittaa tässä samassa ohjelmassa.

Työ vastaa hyvin sille asetettuja vaatimuksia ja reunaehtoja. Näiden vaatimusten määrittely työn alkaessa oli kuitenkin vaikeaa johtuen useista teknisistä seikoista ja asiakkaan ennakkokäsityksistä. Mikäli sisäänkirjautuminen olisi toteutettu perinteisellä tavalla ja käyttäjänimi sekä salasana kombinaatiolla, niin olisi käytettävät ohjelmointikielet ja viitekehykset olleet mahdollisesti erilaisia. Pelin ensimmäinen versio suoritettiin PHP-ohjelmointikieleen vahvasti nojaavalla Nginx-serverillä, jolla edellä mainitun ratkaisun suorittaminen tietokantoineen on hyvin yleistä ja siitä löytyy myös paljon dokumentaatiota.

Prosessi kokonaisuudessaan havainnollistaa myös huomattavissa määrin yleisellä tasolla verkkopohjaisia teknologioita, sekä ohjelmistokehitystä verkkoympäristössä. Suunnitteluvaiheessa ja ensimmäisissä iteraatioissa sosiaalisen median sisäänkirjautumisjärjestelmän toteuttaminen osoittautui teknisellä tasolla vaativammaksi verrattuna ennakko-oletuksiin. Tämän lisäksi on ollut kiinnostava ja opettavainen katsaus JavaScriptin viimeisimpiin sovelluskohteisiin ja ohjelmistokehityksen prosesseihin. Kyseinen kieli on tunnettu aikaisemmin verkkosivujen selaimessa suoritettavana kielenä, joka on lähinnä tuonut lisäominaisuuksia html-koodin manipulointiin. Tänä päivänä sillä voidaan full stack-ratkaisuina tuottaa laajalle yleisölle skaalattavia suorituskyvyltään tehokkaita verkkopalveluita.

Lähteet

Painetut

Pohjonen, R. 2002. Tietojärjestelmien kehittäminen. Jyväskylä: Docendo

Haikala, I., Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Helsinki: Talentum

Juvonen, R. 2018. Ohjelmistoprojektin sudenkuopat ja miten ne vältetään. Helsinki: Books On Demand

Sähköiset

Implement the OAuth 2.0 Authorization Code with PKCE Flow. 2007. Viitattu 01.08.2019. https://oauth.net/about/introduction/

Understanding OAuth2 and Building a Basic Authorization Server of Your Own: A Beginner's Guide. 2018. Viitattu 15.08.2019. https://medium.com/google-cloud/understanding-oauth2-and-building-a-basic-authorization-server-of-your-own-a-beginners-guide-cf7451a16f66

Using template engines with Express. 2019. Viitattu 20.08.2019. https://expressjs.com/en/guide/using-template-engines.html

Kuviot

Kuvio 1: OAuth flow 11

Kuvio 2: Package.json tiedosto, Node.js-liitännäiset 12

Kuvio 3: Lisäosien lisääminen toteutukseen..... 13

Kuvio 4: Express.js 13

Kuvio 5: Juurihakemiston ja portin määrittäminen 14

Kuvio 6: Router() 14

Kuvio 7: Node.js-reitit 15

Kuvio 8: Html-käyttöliittymä testausta varten 15

Kuvio 9: Keys.js 16

Kuvio 10: MongoDB:n implementointi 16

Kuvio 11: MongoDB kirjautumistiedot 17

Kuvio 12: MongoDB yhdistäminen 17

Kuvio 13: 'connected to mongodb' 18

Kuvio 14: Google-strategia 18

Kuvio 15: Tietokantakyselyt call back-funktion sisällä 18