Santeri Hartikainen

LYHYTLINKKIJÄRJESTELMÄN BACKEND-SOVELLUKSEN TOTEUTUS

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittely

2020

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkintonimike	Aika
Santeri Hartikainen	Tradenomi (AMK)	Marraskuu 2020

Opinnäytetyön nimi

Lyhytlinkkijärjestelmän Backend-sovelluksen toteutus — 32 sivua

Toimeksiantaja

Sanoma Pro Oy

Ohjaaja

Janne Turunen

Tiivistelmä

Opinnäytetyön tavoitteena on dokumentoida, miten toimeksiantajalle tehtävä lyhytlinkkijärjestelmän backend-sovellus toteutetaan. Työn tavoite on luoda toimeksiantajalle sovelluksen toimiva backend-versio, joka voidaan yhdistää erikseen tehtävän frontend-sovelluksen kanssa. Aluksi käydään läpi työssä käytettävät tekniikat ja paikoin niiden historiaa, jota seuraa käytännön toteutus. Työssä ei toteuteta sovelluksen frontend:iä.

Työssä käytetään Javascript:in pohjautuvaa Node.js ohjelmointiympäristöä. Lisäksi käytössä on useita sille tehtyjä lisäkirjastoja. Yksi tällainen kirjasto on Express, joka vastaa mm. reitittämisestä ja johon monet muut suositut Node.js kirjastot perustuvat. Sovelluksessa käytettävä tietokanta on SQLite ja sen manuaalisen käsittelyn apuna on DB Browser for SQLite-sovellus. Ohjelman REST-rajapinnan HTTP-kutsujen testaamisessa käytetään Postman-sovellusta.

Koodaus aloitetaan toteuttamalla tietoa hakevat kutsut ja lisäämällä testikäyttäjiä tietokantaan, jonka jälkeen siirrytään toteuttamaan käyttäjien kirjautuminen, heidän kirjautumisensa voimassaolon todentamiseen ja lyhytlinkkien lisäämiseen tietokantaan. Seuraavaksi toteutetaan automatisoidut toiminnot, kuten vanhentuneiden linkkien poisto ja linkin vanhenemisesta varoittavan sekä ilmoittavan sähköpostin lähetys kyseisen linkin omistajalle. Lopuksi toteutetaan käyttäjien ja linkkien hallinta, johon kuuluu käyttäjien ja linkkien muokkaaminen, poisto, sekä uuden salasanan luonti ja sen sähköpostilähetys käyttäjälle, joka on mahdollisesti sen unohtanut.

Lopputuloksena syntyy valmis backend-sovellus, jolle voidaan tehdä API-pyyntöjä ja joihin se vastaa. Päätännössä käydään läpi mitä haasteita työssä oli, sekä mitä tehtäisiin mahdollisesti toisin tulevaisuudessa samankaltaisissa projekteissa, sekä mitä sovellukselle voitaisiin tehdä seuraavaksi ennen sen käyttöönottoa.

Asiasanat

Ohjelmointi, sovelluskehitys, backend-sovellus, REST-arkkitehtuuri

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Santeri Hartikainen	Bachelor of Business Administration	November 2020

Thesis title

Short link application's backend implementation — 32 pages

Commissioned by

Sanoma Pro Oy

Supervisor

Janne Turunen

Abstract

The aim of the thesis was to document how the backend application of the short link system was implemented. The aim of the work was to create a backend version of the application for the client, which could be combined with a separately made frontend application. First, the thesis introduces techniques and history of the techniques that were used in this work, followed by practical implementation. Frontend was not implemented.

The work used the Javascript-based Node.js programming environment. In addition, several additional libraries were in use, such as Express, which corresponds to e.g. routing and which many other libraries are based on. The database in use was SQLite and was managed by DB Browser for SQLite. The program was tested using the application called Postman.

Coding was initiated by executing information retrieval calls and adding test users to the database, followed by user's login, validating their login validity, and adding short links to the database. Next, automated functions were implemented, such as deleting expiring links and sending an email alerting and notifying the link owner. Finally, user and link management were implemented, which included editing and deleting users and links, and creating a new password and emailing it to a user who may have forgotten it.

The result was a ready-made backend application for which API requests could be made and to which it responded. Lastly, the thesis discussed what the challenges were, as well as what might be done differently in the future, and what the application might need next.

Keywords

Programming, software development, back-end application, REST architecture

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön tavoitteena on toteuttaa Sanoma Pro:lle lyhytlinkkijärjestelmän backend käyttäen REST-rakennetta. Työssä ei toteuteta frontendiä. Työn toimeksiantaja, Sanoma Pro on Suomen suurin oppimateriaalikustantaja. Sanoma Pro on osa Sanoma-konsernia. Tavoitteena on, että Sanoma Pro voi käyttää tätä backend-sovellusta erikseen tehtävän selainpään sovelluksen kanssa tehdäkseen lyhennettyjä ja helpommin jaettavia linkkejä firmansa sisällä.

Toteutuksessa käytettiin palvelimen sekä sen REST-puolen toteutukseen Javascriptiin pohjautuvaa Node.js -palvelin ohjelmointiympäristöä, sekä sen lisäkirjastoja, kuten Expressiä, Jsonwebtokenia sekä Nodemaileria. Tietokantana käytettiin SQLiteä.

Luvussa kaksi käydään läpi mitä tekniikoita toteutettavassa työssä käytetään. Aloitetaan käymällä läpi, mikä Node.js on, jonka jälkeen käydään läpi, miten sitä ja sen lisäkirjastoja työssä käytetään. Tämän jälkeen selvitetään, mikä työssä käytettävä REST-arkkitehtuuri on, jonka jälkeen selvitetään, mitä tietokantamoottoria käytetään ja mitkä sen perustoiminnallisuudet ovat. Viimeiseksi käydään läpi, mikä kirjautumisen tunnistautumisessa käytettävä JSON-webtoken on.

Luvussa kolme käydään läpi työn toteutus. Ensimmäisenä selvitetään, miten Node.js sovelluksen teko aloitetaan ja miten sillä saadaan tehtyä tietokannasta dataa hakevia kutsuja käyttäen Postman-sovellusta, jonka jälkeen toteutetaan käyttäjien lisääminen, kirjautuminen, sekä itse lyhytlinkkien teko. Lopuksi toteutetaan automatisoidut toiminnot, linkkien muokkaus, niiden poisto ja käyttäjien hallinta.

Työssä käytetään englannin kieltä muuttujien ja funktioiden nimeämisessä. Tähän päädyttiin, koska jos sovellusta tarvitsee joskus mahdollisesti korjata tai muokata, niin onnistuu se helposti myös suomen kieltä taitamattomalta henkilöltä.

2 TYÖSSÄ KÄYTETTÄVÄT TEKNIIKAT

2.1 Node.js

Node.js tai lyhyemmin Node on Javascriptiin perustuva tapahtumapohjainen verkkosivujen palvelinpään ohjelmointiteknologia (Tsonev 2014, 27). Sen selvimpiä vahvuuksia ovat nopeus prosessointi ajassa. Sen suosiota lisää myös se, ettei ohjelmoijan tarvitse käytännössä osata verkkosovellusta tehdessään muita kieliä kuin Javascriptiä.

Noden ensimmäisen version kehitti vuonna 2009 Googlella työskennellyt Ryan Dahl, joka oli ollut tyytymätön silloisten palvelinten kykyyn käsitellä useita (yli kymmeniä tuhansia) samanaikaisia käyttäjiä ja halusi mahdollistaa Google uuden Chrome-selaimen Javascript V8 tulkin käytön verkkosivujen palvelimen käytössä (Sharma 2017). Javascriptiin pohjautuvaa palvelin tekniikkaa oli kylläkin kehitelty jo kolmetoistavuotta aiemmin Netscape nimisen verkkoselaimen yhteydessä, mutta vasta Noden kehitys nosti Javascriptin näkyville palvelinpään ohjelmointi kehityksessä. Dahl jatkoi Noden kehitystä vuoteen 2012 asti, jolloin hän päätti luovuttaa Noden kehitysvastuun sen npm kirjaston kehittäjän Isaac Sculueterin harteille, jotta hän voisi itse keskittyä työskentelyyn omien tutkimustöidensä parissa. (O'Dell 2012.)

Node.js käyttää yksisäietekniikkaa (eng, Single thread technology). Toisin kuin muiden paljon käytettyjen palvelinkielten, kuten Javan käyttämässä monisäietekniikassa (eng, Multi thread technology), Node.js käyttää prosessien käsittelyyn vain yhtä säiettä siinä missä monisäietekniikkaa käyttävä ohjelma ottaa käyttöön aina uuden säikeen jokaiselle uudelle kutsulle (Tsonev 2014, 27). Noden tapahtumasilmukka on tehty niin, ettei se jää odottamaan vastaamattomia kutsuja, vaan jatkaa suorittamista eteenpäin. Tämä helpottaa Nodella tehtävien sovellusten kehitystä, sillä sovelluskehittäjän ei juurikaan tarvitse varoa esimerkiksi hitaan tai epäonnistuneen tiedostonlukemisen tuottamaa palvelimen jumittumista. (Node.js s.a.) Tästä voi toki syntyä ongelma, jos esimerkiksi tietokannasta tarvittavaa tietoa ei keretä hakemaan ennen kuin ohjelma sitä tarvitsisi. Nodelle on tämän ongelman ratkaisuksi kehitetty kirjastoja, jotka jäävät odottamaan haluttuja kutsuja.

Nodelle on tehty todella paljon kirjastoja, jotka helpottavat sillä tehtävien sovellusten tekoa. Tällaisia kirjastoja ovat mm. Express, Cors, sekä Cookie-parser. Näiden kirjastojen hallintaan on kehitetty Npm-niminen järjestelmä, jolla sovelluskehittäjä voi komentorivin kautta asentaa ja julkaista helposti lisäkirjastoja. Npm luotiin vuonna 2009 aluksi pelkkänä avoimen lähdekoodin projektina, jonka tarkoitus oli helpottaa koodin jakoa Javascriptiä käyttävien ohjelmoijien kesken. Vuonna 2014 perustettu npm, inc. otti tehtäväkseen Isaac Sculueterin johdolla Npm-paketinhallinnan ja sen kehityksen. He ovat keskittäneet resurssinsa Npm:n ilmaisena avoimen lähdekoodin -palveluna pitämiseen ja rakentaakseen tietoturvallisia työkaluja yrityksille ja yksityisille käyttäjille. He ovat myös luoneet maksullisia palveluita, joiden tarkoitus on auttaa päivittäin Javascriptillä työskentelevien ohjelmoijien työtä. (Npm s.a.a.)

Ohjelmointiin perehtyneelle henkilölle yksinkertaisen paikallisen palvelinsovelluksen (kuva 1.) tekeminen Nodella on melko helppoa. Aloitetaan luomalla palvelin.js niminen tiedosto. Kun tiedosto on luotu, otetaan http-moduuli käyttöön ja tallennetaan se objektiin, jonka kautta tämän moduulin ominaisuuksia päästään helposti hyödyntämään. Palvelimelle tulee määrittää osoite, joka paikallisissa testailuissa on 127.0.0.1, sekä vapaasti määriteltävä portti. Nyt voidaan kutsua http-moduulin createServer-metodia ja tallentaa sen tiedot ”palvelin” -objektiin. Tälle createServer-metodille voidaan antaa parametreiksi pyyntö req ja vastaus res. Pyyntöön voidaan ottaa vastaan käyttäjän tietoja, joita funktio voi sitten käyttää. Vastaus taas tulostaa käyttäjälle palautteen, joka voi olla esimerkiksi tekstiä. Palvelin saadaan päälle http-moduulin listen-funktiolla. Sille täytyy vain antaa aiemmin määritelty portti, sekä osoite, jonka jälkeen voidaan komentorivillä siirtyä tiedoston alle ja käynnistää palvelin komennolla ”node palvelin.js”. Tämän jälkeen, jos mennään selaimella osoitteeseen 127.0.0.1:3000, pitäisi sivulla olla tulostettuna palvelinobjektin res.end komennon viesti: ”Yksinkertainen Node.js esimerkki!”. (Kuva 1.)

```
const http = require('http');
const osoite = '127.0.0.1';
const portti = 3000;

const palvelin = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Yksinkertainen Node.js esimerkki!');
});

palvelin.listen(portti, osoite, () => {
  console.log(`Palvelin päällä osoitteessa http://${osoite}:${portti}/`);
});
```

Kuva 1. Yksinkertainen paikallinen Node.js palvelin sovellus

Yksi Noden hyötyjä on sillä tehtävien moduulien helppokäyttöisyys. Esimerkkinä voitaisiin tehdä moduuli, joka palauttaa tämänhetkisen ajan. Moduulit kannattaa selkeyden vuoksi jakaa usein omiin tiedostoihinsa. Aloitetaan luomalla tiedosto nimellä moduuli.js. Mooduuli.js-tiedostoon tehdään kellonaikafunktio. Funktion alkuun lisätään export, jotta funktiota voitaisiin käyttää tämän tiedoston ulkopuolella. Sidotaan muuttujaan paiva tämän hetkisen päivän tiedot Javascriptin new Date()-komennon avulla. Irrotetaan paivasta halutut tunnit ja minuutit omiin muuttujiin paiva.getHours()-funktion ja paiva.getMinutes()-funktion avulla. Laitetaan moduuli palauttamaan tiedot return-palautuksen mukana muodossa tunti + ”:” + minuutit. (Kuva 2.)

```
exports.kellonaika = function () {
    let paiva = new Date();
    let tunti = paiva.getHours();
    let minuutit = paiva.getMinutes();
    return tunti + ":" + minuutit;
};
```

Kuva 2. Kellonaika moduuli

.

palvelin.js-tiedostossa pitää määritellä mistä haluttu moduuli löytyy. Kun kummatkin tiedostot ovat samassa kansiossa, voidaan moduuli asettaa muuttujalle kirjoittamalla: require(’./moduuli’). Nyt palvelin objektin res.end viestiin voidaan lisätä nykyinen kellonaika käyttämällä moduulia kirjoittamalla: moduuli.kellonaika(). (Kuva 3.)

```
const http = require('http');
const osoite = "127.0.0.1";
const portti = 3000;
const moduuli = require('./moduuli');

const palvelin = http.createServer(function (req, res) {
    res.end('Kello  on: ' + moduuli.kellonaika());
});

palvelin.listen(portti, osoite, () => {
    console.log(`Palvelin päällä osoitteessa http://${osoite}:${portti}/`)
})
```

Kuva 3. Moduulia käyttävä palvelin

Nodea käytetään hyvin harvoin ilman sille tehtyjä lisäosia. Yksi suosituimmista lisäosista on Express, jolla voidaan helpottaa sovelluksella tehtäviä kutsuja ja johon monet muut Noden suositut lisäosat pohjautuvat (Express s.a.a.).

2.2 Express

Express on avoimenlähdekoodin Node.js kirjasto, jonka päätehtävä on helpottaa verkkosivujen eri http-kutsujen käsittelyä. Sen avulla voi helposti tehdä palvelinsovelluksia, joille voi asettaa erilaisia tapoja vastata eri URL-osoitteista tuleviin kutsuihin. (Vivah 2017.) Tämä reitittäminen tapahtuu väliohjelmistojen (middleware) avulla. Väliohjelmistot ovat funktioita, jotka saavat palvelimelle tulevia pyyntöjä (request), muokkaavat lähteviä vastauksia (response), sekä kutsuvat muita funktioita. Expressin väliohjelmistot voidaan jakaa 5 eri tasoon. Nämä tasot ovat sovellustaso, reititystaso, virheenkäsittely, sisäänrakennettu-taso, sekä kolmannen osapuolentaso. (Express s.a.b.)

Sovellustason väliohjelmistolla voidaan reitittää, hallita väliohjelmistoja ja tulostaa HTML-näkymiä käyttäjälle. Sen ominaisuudet saadaan käyttöön sitomalla express()-ilmentymä muuttujaan. Reititystaso on sovellustason alataso, jonka ainut tehtävä on reitittäminen. Se saadaan käyttöön sitomalla ilmentymä express.Router() haluttuun muuttujaan. Virheenkäsittely väliohjelmistolla voidaan napata ajon aikana tapahtuvia virheitä ja määrittää mitä niiden sattuessa tehdään. Funktiolle tulee määritellä argumenteiksi pyyntö, vastaus ja halutessa next-komento. Jos halutaan saada virheenkäsittely mukaan, täytyy näiden 3 argumentin lisäksi määritellä ensimmäisenä virheargumentti, jolla tulleen virheen tietoihin päästään käsiksi. Sisäänrakennettuja väliohjelmistoja

ovat sivun staattisia osia hallinnoiva express.static, JSON tiedostojen käsittelyyn express.json ja express.urlencoded, jolla käsitellään koodikieleksi muutettuja URL:leja. Kolmannen osapuolen väliohjelmistoja saa ladattua Noden kirjastonhallintatyökalun npm:n kautta. (Express s.a.b.)

Edellisen kappaleen kellonaika demon toteutuksessa voitaisiin hyödyntää Expressiä seuraavanlaisesti. Aloitetaan luomalla uusi, esimerkiksi palvelin.js-niminen Javascript-tiedosto omaan kansioonsa. Kun komentorivillä on navigoitu tähän kansioon, kirjoitetaan alustuskomento npm init. Komentoriville tulee komentoja, joilla luodaan JSON tiedosto, johon voidaan lisätä tämän tiedoston halutut tiedot. Tässä demossa voidaan hyvin käyttää vakio tietoja, joista tulee vain varmistua, että entry point on sama kuin käytetty Javascript-tiedosto. Tämän jälkeen tulee express asentaa tähän projektiin komennolla npm install express –save.

Nyt kun alustukset on tehty, voidaan kopioida edellisen kappaleen demo ja lähteä muokkaamaan sitä. Aluksi tuodaan Express-muuttujaan express tutulla require-komennolla. Edellisestä demosta poiketen, jossa palvelin-muuttujaan sidottiin koko http.createServer-funktio, nyt palvelin-muuttujaan sidotaan pelkkä express. Nyt expressiä voidaan käyttää eri reittien REST pyynnöissä. Reitti tehdään käyttämällä palvelin-muuttujaa. Tässä esimerkissä halutaan vain saada tietoja palvelimelta, joten käytetään pelkkää palvelin.get pyyntöä. Tämä tarvitsee toimiakseen kolme parametriä. Ensimmäinen on polku, mihin pyyntö tehdään. 127.0.0.1:3000 on palvelimen juuri ja sen polku on pelkkä /. Jos kuitenkin halutaan toimintoja ulkoistaa omiin polkuihinsa, kuten tässä demossa 127.0.0.1:3000/kellonaika polkuun, onnistuu se yksinkertaisesti asettamalla get-pyynnön poluksi kellonaika. (Kuva 4.)

Kun polku on määritelty, tarvitsee pyyntö request ja response parametrit. Res.send lähettää selaimelle halutut tiedot ja siihen voidaan lisätä aiemmin tehty kellonaikamoduuli (kuva 4). Nyt kun selaimella mennään osoitteeseen 127.0.0.1:3000/kellonaika, tulostuu sinne tämänhetkinen kellonaika.

```
const express = require('express')
const osoite = "127.0.0.1";
const portti = 3000;
const moduuli = require('./moduuli');
const palvelin = express();

palvelin.get('/', (req, res) => {
  res.send('Kellon aika demo')
})
palvelin.get('/kellonaika', (req, res) => {
    res.send('Kello  on: ' + moduuli.kellonaika())
})
palvelin.listen(portti, osoite, () => {
    console.log(`Palvelin päällä osoitteessa http://${osoite}:${portti}/`)
})
```

Kuva 4. Moduulia käyttävä express palvelin

Näin yksinkertaisessa esimerkissä voi olla vaikea huomata, miten paljon Express auttaa Nodella tehtävien sovellusten tekemistä. Expressin hyödyt alkavat tulla esille, kun palvelimelle aletaan asettaa useita eri polkuja.

2.3 REST-arkkitehtuuri

REST lyhentyy sanoista Representational State Transfer. Sillä tarkoitetaan vakioitua rakenteellista tyyliä, jonka tehtävä on helpottaa internettiä käyttävien laitteiden kommunikointia keskenään (Agile Education Research 2019). REST:n suurin etu on se, ettei palvelinpään ja käyttäjäpään ei tarvitse olla tietoisia toisistaan ja riittää, että kummatkin puolet käyttävät samaa kommunikointirakennetta. Palvelimeen ja käyttäjäpäähän tehtävät muutokset eivät täten vaikuta toinen toisiinsa. Näin voidaan rakentaa laajoja ja joustavia palveluita, joiden eri osien toteutuksia voidaan jakaa helposti useille eri tekijöille. Palvelinten ja laitteiden, jotka käyttävät REST-rakennetta ei tarvitse olla millään tavalla tietoisia toistensa tilasta. Ne myös ymmärtävät toinen toisiaan ilman tietoa mahdollisista edeltäneistä kutsuista. (Codecademy s.a.)

REST-rakenteessa käyttäjäpää lähettää pyyntöjä palvelimelle. Saatuaan pyynnön palvelin, jolta tietoa haetaan, joko palauttaa vastauksen selaimelle tai muuttaa itsessään olevia tietoja pyynnön mukana tulevan datan mukaan. Tällaiseen palvelimeen viitataan usein käyttämällä termiä API (application programming interface). (Codecademy s.a.) Resursseja käsitellään HTTP-proto-

kollan metodeilla, jotka määrittelevät millaisesta kutsusta on kyse. HTTP-protokollan metodeja ovat GET, POST, PUT ja DELETE. GET verbillä selaimelle palautetaan tietoa, POST lisää palvelimelle tietoa, PUT muokkaa palvelimella olevaa tietoa ja DELETE poistaa palvelimella olevaa tietoa (Agile Education Research 2019).

REST pyyntöihin voidaan sisällyttää header, johon käyttäjäpää voi lisätä, millaista tietoja se voi ottaa vastaan. Tällä vältetään tilanteita, joissa palvelin lähettää selaimelle tietoa, jota se ei osaa tulkita. Tähän accept-kenttään lisätään halutun tiedon tyyppi, sekä kauttamerkillä erotettu alatyyppi. Esimerkkinä yleistekstistä olisi malli text/plain. Jos taas halutaan tekstitiedosto, johon on sisällytetty CSS elementtejä, niin olisi muoto text/css. Jos haetaan media tiedostoja, niin laitetaan niiden header osioon ensiksi tiedontyyppi, jota haetaan ja alatiedoksi mikä tiedostopääte tiedostolla on. (Codecademy s.a.)

Esimerkiksi jos kyseessä olisi verkkosivun artikkeli API, josta haluttaisiin saada png-muotoinen valokuva artikkelista, tulisi accept-kenttään antaa arvoksi image/png (kuva 5).

Kuva 5. GET-pyyntö

HTTP pyyntö tarvitsee tiedon hakuun polun, josta tietoa haetaan ja mitä HTTP-metodia käytetään (kuva 6).

HTTP metodi	Osoitteen loppuosa	Esimerkki url: https://api.esimerkki.com/data
GET	/kirjat	Hakee kaikien kirjojen tieodot vastauksen body-elementtiin.
POST	/kirjat	Luo osoitteeseen /kirjat uuden kirjan pyynnössä olevan datan perusteella.
PUT	/kirjat{id}	Muokaa osoitteen /kirjat kirjaa, jonka id tunniste vastaa pyynnön id tunnistetta.
DELETE	/kirjat{id}	Poista osoitteen /kirjat kirjan, jonka id tunniste vastaa pyynnön id tunnistetta.

Kuva 6. HTTP-metodit

Kun palvelin lähettää tietoa selaimelle, on sen lisättävä content-type-osio vastauksen header-osioon. Tämä kertoo selaimelle, millaista tietoa on tulossa. Palvelin myös lähettää samassa yhteydessä vastauskoodin, josta ilmenee, onnistuiko tiedon lähetys. Eri vastauskoodeja on useita, mutta tärkeimmät koodit ovat: 200 = OK, 404 = tietoa ei löydy ja 500 = yleinen palvelimenvirhe. (Codecademy s.a.) REST-rakennetta voidaan käyttää haettaessa tietoa esimerkiksi SQLite tietokannasta.

2.4 SQLite

SQLite on C-ohjelmointikielellä toteutettu SQL-tietokantamoottori. Se on ilmainen ja vapaasti käytettävissä kenelle tahansa. Sen kehitys aloitettiin vuonna 2000 ja sen tukea on suunniteltu jatkettavaksi vuoteen 2050 asti. Sen vahvuuksia ovat sen nopeus, pieni koko ja ulkoisten kirjastojen vähyys. (SQLite s.a.a.) SQLite ei ole riippuvainen käyttöjärjestelmästä eikä sillä ei ole juurikaan riippuvuuksia esimerkiksi ulkoisista kirjastoista muutamia standardi C-ohjelmointikielen kirjastoja lukuun ottamatta (SQLite s.a.b.). Monet suuret yritykset käyttävät SQLite:ä laitteissaan ja palveluissaan. Mm. Android ja IOS laitteet käyttävät SQLite:ä lähes kaikissa natiivisovelluksissaan. (SQLite s.a.c.)

Toisin kuin monissa muissa tietokantamoottoreissa, ei SQLite tarvitse erillistä palvelinta toimiakseen. Se lukee ja kirjoittaa tietokannan tiedot suoraan normaalille levylle. Hyötynä tästä on se, ettei se tarvitse erillistä hallinnointia tietokannan luontivaiheessa. Jos sovellus pääsee käsiksi levyllä oleviin tietoihin,

niin pääsee se myös käsiksi tietokantaan. Näin sovellus ei tarvitse tietokannan käyttöön internettiä. Kääntöpuolena palvelimella olevat tietokannat eivät altistu yhtä herkästi käyttäjäpäästä johtuviin ongelmiin kuin palvelimettomat tietokannat. (SQLite s.a.d.)

SQLiten komennot seuraavat SQL-standardia (SQLite tutorial s.a.b.). SQLite taulukko saadaan luotua CREATE TABLE-komennolla, jonka jälkeen lisätään hakasulkuihin lisätieto IF NOT EXISTS. Tämä luo uuden taulukon, mikäli saman nimistä taulukkoa ei löydy. Taulukon nimi sekä missä tietokannassa se sijaitsee, määritellään äskeisen komennon perään. (SQLite tutorial s.a.a.) Tässä esimerkissä tietokannan nimi, joka kirjoitetaan hakasulkeisiin, on tietokanta ja taulukko nimi, joka taas tulee hakasulkeiden perään pisteellä, on henkilo. Näiden jälkeen voidaan lisätä sulkuihin sarakkeita, joita taulukko sisältää.

Sarakkeille tulee ensiksi asettaa niille yksilöllinen nimi. Tämän jälkeen sille määritellään datamuoto ja sarake rajoitukset. Datan muoto voi olla esimerkiksi TEXT eli tekstiä tai INT eli tasaluku. Sarakkeen rajoituksiin voidaan asettaa rajoituksia kuten PRIMARY KEY, DEFAULT, sekä NOT NULL. PRIMARY KEY asettaa sarakkeen sen taulukon yksilöiväksi tiedoksi. NOT NULL pysäyttää sarakkeen lisäämisen, mikäli käyttäjä ei anna sille arvoa. (SQLite tutorial s.a.a.) DEFAULT asettaa sarakkeelle sen perään asetetun arvon, mikäli käyttäjä ei saraketietoja lisätessään anna sille mitään arvoa (W3Schools s.a).

```
CREATE TABLE [IF NOT EXISTS] [tietokanta].henkilo (
    ID INT PRIMARY KEY,
    Nimi TEXT NOT NULL,
    Ika INT DEFAULT 0
);
```

Kuva 7. SQLite CREATE TABLE

Yksi käytetyimmistä SQL komennoista on SELECT-komento, jolla saadaan haettua tietoa SQL-tietokannasta. SQLitessä tämä komento toimii samoin kuin SQL-standardissa. SELECT-komennolla haetaan tietoa yhdestä tai useammasta sarakkeesta. (SQLite tutorial s.a.b.) Jos lähdettäisiin hakemaan tietoja tietokannasta, jonka rakenne olisi äsken tehdyn taulukon luontiesimerkin mukainen, onnistuisi se seuraavanlaisesti. Määritellään, että haetaan sarakkeen kaikki tiedot kirjoittamalla * (astralis) heti SELECT-alkuosan perään. Tämän

jälkeen lisätään mistä tätä tietoa haetaan kirjoittamalla FROM ja taulukon nimi, joka tässä tapauksessa on ”henkilo”, jonka perään voidaan lisätä WHERE, jolla saadaan rajattua tietoa. Rajaavaksi tiedoksi valitaan usein ID. Tämä SQL-lause näyttäisi tältä: SELECT * FROM henkilo WHERE ID = 1.
Tällä komennolla saadaan tietokannasta henkilo-taulukon sarake, jonka ID on 1.

2.5 JSON Web Token

JSON Web Token, lyhyesti JWT, mahdollistaa tietojen välityksen eri tahojen välillä JSON-objektimuodossa. JWT ns. allekirjoitetaan joko käyttäen algoritmin avulla tehtävää salausta tai käyttäen avainvertailua. Tämän ansiosta lähetettävä tieto voidaan todentaa ja luottaa siihen, että tiedon lähettäjä sekä vastaanottaja ovat tiedon käsittelyyn valtuutettuja tahoja. (JWT s.a.)

JWT on hyödyllinen apuväline todennettaessa käyttäjän valtuuksia. Kun käyttäjä on kirjautunut, tehdään jokaisen kutsun yhteydessä tarkistus käyttäjän token:n voimassaolosta sekä siitä, onko kyseisen token:n omistaja sallittu pääsemään tiettyyn polkuun. Kun käyttäjä tekee pyynnön suojattuun polkuun, tulee käyttäjäpään lähettää luvan anto header, johon kuuluu JWT token. Kun JWT sisältää tarvittavat tiedot, päästetään käyttäjä käyttämään polusta löytyvää dataa. (JWT s.a.)

JWT rakenteeseen kuuluu header, johon sisältyy tieto token-muodosta ja allekirjoituksen algoritmimuoto. Header:n lisäksi rakenteeseen kuuluu ns. kuorma, johon voidaan lisätä tietoa yleisesti käyttäjästä, sekä muita haluttuja lisätietoja. Viimeinen rakenteen osanen on allekirjoitus, johon yhdistetään koodikieleksi muutettu header ja kuorma, salaisuus ja salauksen algoritmimuoto. (JWT s.a.)
JWT token:t ajastetaan vanhenemaan tietyssä ajassa. Vanhenemisaika voidaan asettaa millisekunneista ylöspäin. (Npm s.a.b.)

JWT on varteenotettava vaihtoehto verrattaessa sitä muihin token-muotoihin. Näistä käytetyimpiä ovat Simpe Web Token (SWT) ja Security Assertion Markup Language Tokens (SAML). Koska JWT on tehty käyttäen JSON tiedostomuotoa, on se huomattavasti kompaktimpi kuin SAML, joka perustuu XML-

merkkistandardiin. JWT on myös verrattain turvallisempi SWT:hen, sillä JWT mahdollistaa useamman kuin yhden tavan salata tietoa. (JWT s.a.)

3 SOVELLUKSEN TOTEUTUS

3.1 Projektin aloitus

Node.js on valmisteltava projektin alussa kirjoittamalla komentoriville npm init. Komentorivillä voidaan antaa projektille oletustiedot. Tämä luo kansioon package.json-tiedoston, jonka jälkeen asennetaan Express npm-komennolla, joka lataa Expressin tarvitsemat Node-moduulitiedostot.

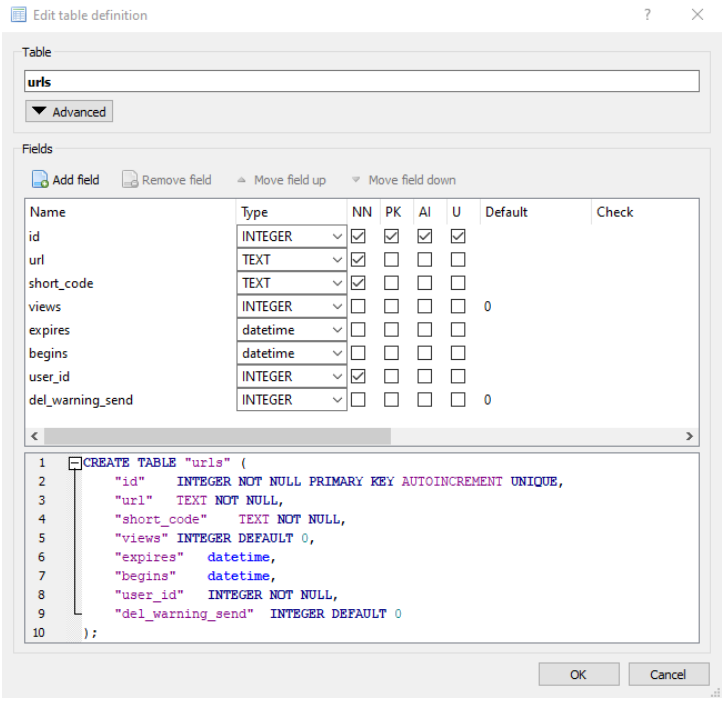
Seuraavaksi luodaan index.js tiedosto työn juureen ja lisätään sinne perinteinen Node-runko, jotta palvelinta pystytään ajamaan paikallisesti portissa 3000 (kuva 8). Tämän jälkeen asetetaan juureen tulevat kutsut käyttämään routes kansion urls-tiedostoa app.use-komennolla. Juureen voidaan lisätä valmiiksi kansio routes tulevia moduuleja varten, joihin tullaan ulkoistamaan tiettyjä toimintoja ja luomaan sinne tiedostoja, jotka käsittelevät polkujen kutsuja. Lisäksi on hyvä tehdä views-kansio, jonne voi lisätä näkymiä hallitsevia pug-tiedostoja, sekä db nimisen kansion, jonne tietokannat sijoitetaan.

```
const express = require('express');
const app = express();
const adress = "127.0.0.1";
const port = 3000;

var indexRouter = require('./routes/urls');

app.set('view engine', 'pug');

app.use('/urls', indexRouter);

app.listen(port,adress, () => {
  console.log(`palvelin päällä osoitteessa : http://localhost:${port}`)
})
module.exports = app;
```

Kuva 8. Node.js runko

Kun projektiin on lisätty pug-riippuvuudet npm-komennolla, voidaan lisätä itse pug-tiedosto tarpeellisilla tiedoilla, jolla helpotetaan testausvaiheen toiminnallisuuksien testausta, sekä vältetään näkymän puuttumisesta johtuvien turhien virheilmoitusten syntymistä. Seuraava valmistelujen askel on luoda tietokanta, jonne tullaan tallentamaan sovelluksen käyttämä data.

SQLite tiedostojen kehitysvaiheen käyttöä varten on hyvä ladata DB browser for SQLite, jolla voidaan hallinnoida tietokantaa manuaalisesti. Kun tietokanta tiedosto on luotu, voidaan sinne tehdä urls- ja users- taulukot create table sarakkeessa. (Kuva 9).



Kuva 9. Urls-taulukon määrittely

Ennen kuin ohjelma on saatu siihen pisteeseen, että sillä voidaan lisätä tietoja tietokantaan, voidaan sinne lisätä dataa execute SQL-sarakkeessa SQL-komennoilla. Kuitenkin, jotta palvelin pystyisi edes lukemaan tietokantaa, täytyy projektiin asentaa sqlite3-moduuli komentorivillä.

3.2 Tietoa hakevat kutsut

Tietokannan tietoja tullaan käsittelemään REST pyynnöillä. Tässä työssä REST pyyntöjen lähettämiseen käytetään ilmaista Postman-sovellusta. Jotta sovelluksella voidaan lähteä tekemään kutsuja, täytyy sen käyttämät toiminnallisuudet luoda palvelinsovellukseen.

Usein halutaan, että REST palvelimet tekevät erilaisia toimintoja, kun niiden eri URL:eja kutsutaan. Tämä onnistuu helpoiten expressin router-väliohjelmiston avulla. Sillä voidaan ohjata sovellus tekemään eri funktiota samasta osoitteesta HTTP-metodin, sekä polun perusteella. (Kuva 10.)

```
router.get('/',CheckShortCode ,function(req, res, next) {
  res.end();
});
```

Kuva 10. Router ohjaus

Seuraavaksi on tehtävä funktio, jota kutsutaan, kun palvelimelle lähetetään get-pyyntö osoitteeseen, joka ohjaa selaimen uuteen osoitteeseen. Julkaistavassa versiossa osoitteen alkuosa voisi olla ”sano/ma?”, jonka perään uniikki lyhytlinkin loppuosa tulee. Kun selain hakee tätä polkua, irrottaa sovellus tämän uniikin loppuosan ja etsii tietokannasta tämän vastakappale osoitteen, jonka jälkeen selain ohjataan tähän osoitteeseen. Testailtaessa kuitenkin lokaalisti on linkin alkuosa ”http://localhost:3000/ma?”.

Tämä kutsu käynnistää async-funktion, jonka sisään voidaan lisätä odotettavia await-funktioita ja jonka parametrinä on tämä lyhytlinkki. Tämä funktio kutsuu getData-funktiota, jonka vastausta odotetaan ennen ajon jatkoa. Funktiolle annetaan haettava merkkijono, joka saadaan hakemalla pyynnön URL ja ottamalla siitä pois sen alkuosa. (Kuva 11).

```
let url = req.url;
let direction = null;
let shortcode = url.substring(2, url.length);
direction = await getData(shortcode);
```

Kuva 11. Async-funktio

Koska halutaan, että getData-funktion vastausta odotetaan, voidaan se sitoa promise:n sisään, joka palauttaa onnistuneesta kutsusta resolve:n ja epäonnistuneesta reject:n. SQLite tietokanta sidotaan muuttujaan, jonka jälkeen se voidaan lukea läpi each-komennolla. Kun haku onnistuu, tarkistetaan, että lyhytlinkin alkamispäivä on astunut voimaan, ja palautetaan resolve:n mukana lyhytlinkkiä vastaava linkki CheckShortCode-funkiolle, joka jatkaa tämän jälkeen ajoa. Tietokannan luku tulee lopettaa close()-komennolla. (Kuva 12.)

```
async function getData(short_code){
  return new Promise((resolve, reject) => {
    let timeNow = new Date().getTime();
    let desired_Url;
    const db = new sqlite3.Database('db/tietokanta.db');
    try{
    db.each(`SELECT * FROM urls WHERE short_code = '${short_code}'`, (err, urls) => {
      if(err) {
        console.error(err.message);
      }
      else{
        urls.begins = new Date(urls.begins).getTime();
        if(timeNow > urls.begins){
          desired_Url = urls.url;
        }else{
          desired_Url = null;
        }
      }
    }, (err,n) => {
        if(err) {
          reject(err);
        }
        else{
          console.log(desired_Url)
          resolve(desired_Url);
        }
        db.close();
      });
      return resolve;
    }catch(err){
      console.log("Error");
    }
  });
}
```

Kuva 12. getData-funktio

Jos getData-funktio palauttaa osoitteen, lisää sovellus tietokantaan kyseiselle linkille yhden käyttökerran. Tämä tapahtuu lähes getData-funktion kanssa identtisellä await-funktiolla, jossa SELECT on muutettu UPDATE muotoon. Tämän jälkeen ohjelma uudelleenohjaa selaimen saatuun osoitteeseen. (Kuva 13.)

```
if(direction != null){
  await AddViews(shortcode);
  res.redirect(direction)
}
res.end();
```

Kuva 13. Uudelleen ohjaus

Vaikka usein halutaan, että lyhytlinkkejä pystyisi käyttämään ilman kirjautumista, ei niiden tekoa haluta välttämättä mahdollistaa kaikille. Tämän takia on sovellukselle tehtävä kirjautumisia, sekä rekisteröintejä varten käyttäjäpuoli.

3.3 Käyttäjän lisääminen

Ensimmäisenä käyttäjäpuolen toiminnallisuutena on hyvä toteuttaa käyttäjien lisäämisen tietokantaan. Lisätään aluksi tarvittavat moduulit Bcrypt, jolla hoidetaan salasanojen salaaminen, Bodyparser, jolla POST pyynnöistä saadaan

luettua rungon mukana tulevat tiedot sekä Cors, jolla vältetään corspolicy:stä johtuvia hakuvirheitä. On myös hyvä luoda erilliset luokat, joiden sisälle HTTP-metodi kutsujen käynnistävät funktiot tulevat. Tällä säästetään rivejä ja selkeytetään koodia, kun routerin HTTP-kutsuihin tarvitsee kirjoittaa pelkkä polku, handler ja funktio, mitä kutsutaan. Lisäksi tietyt funktiot ovat helpommin löydettävissä, kun niiden luokka vain tiedetään.

Käyttäjiä lisäävä funktio saa pyynnön mukana käyttäjänimen, salasanan, sekä sähköpostin, jotka tallennetaan tietokantaan. Salasana salataan Bcrypt-moduulin hash-komennolla. Jotta salasanoja voidaan käsitellä turvallisesti, tulee ne muuttaa muotoon, jota ihminen ei kykene tunnistamaan. Komentoon lisätään vapaasti määriteltävä kierrosten määrä, joka määrää monestiko salasana sotketaan. (Kuva 14.)

```
const saltRounds = 10;
const passwordHash = await bcrypt.hash(password, saltRounds)
```

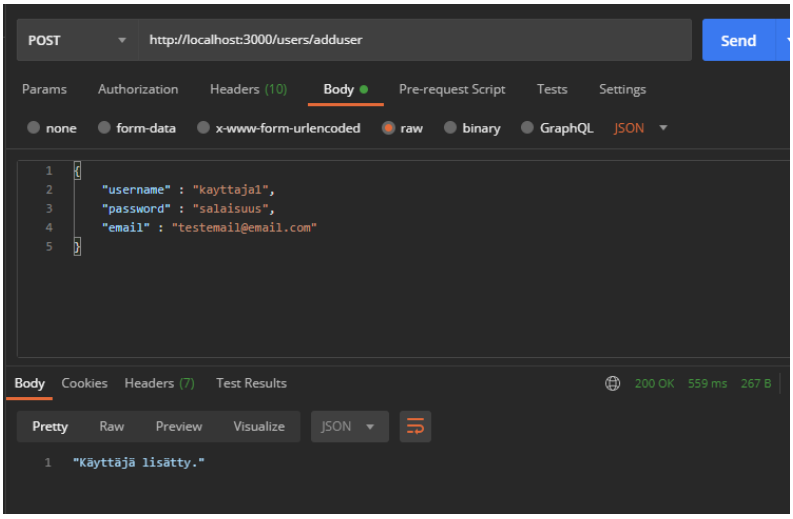
Kuva 14. Salasanan salaus

Tietokanta valmistellaan prepare-komennolla, jossa sille annetaan runko, minne ja miten tietoa lisätään. Lisäys ajetaan run-komennolla, jonka mukana annetaan lisättävät tiedot. Tietokantaa lukeviin funktioihin on paikoin syytä lisätä SQLiten busy_timeout, jotta lisäyksen hitaudesta johtuvaa lisäyksen epäonnistumisilta vältyttäisiin. (Kuva 15.)

```
        db.run('PRAGMA busy_timeout = 10000');
        db.configure("busyTimeout", 10000);

        let add = db.prepare("INSERT INTO users VALUES(?,?,?,?,?)", (err) =>{
        if(err){
            return res.json("Error: " + err);
        }else{
            return res.json("Käyttäjä lisätty.")
        }
        });
        add.run(id, username, passwordHash, email, role);
    }
}
```

Kuva 15. Käyttäjän lisäys

Postman-sovelluksella voidaan nyt lisätä POST-pyynnöllä käyttäjä, jonka rungon mukana on testikäyttäjän tiedot. Rungon tiedot ovat JSON muodossa (kuva 16).



Kuva 16. Käyttäjän lisäys Postman-sovelluksessa

Kutsun onnistuttua tietokantaan on tullut uusi käyttäjä, jonka salasana on salattu. Seuraavaksi toteutetaan kirjautuminen, joka vaaditaan, jos halutaan lisätä lyhytlinkkejä tietokantaan.

3.4 Käyttäjän todennus ja lyhytlinkkien lisääminen

Käyttäjän tunnistamiseksi voidaan ottaa käyttöön Jsonwebtoken-moduuli, jolla voidaan tallentaa käyttäjän todennuksen eli authentikaation voimassa olo. Kirjautuessa funktio saa rungon mukana annetun nimen ja salasanan. Kun käyttäjänimelle löytyy vastakappale, lähdetään vertaamaan tämän käyttäjänimen salattua salasanaa, sekä annettua salasanaa Bcrypt:n compare-komennon avulla. Kun salasanat vastaavat toisiaan, allekirjoitetaan Jsonwebtoken:lla yhden tunnin voimassa oleva token (”tunnus”), joka yhdistetään annettuun käyttäjänimeen ja erillisetä tiedostosta tulevilla sercret:llä Selainpäälle palautetaan itse token, joka voidaan tallentaa jatkon todennusta vaativia toimintoja varten. (Kuva 17.)

```
const db = new sqlite3.Database('db/tietokanta.db',sqlite3.OPEN_READWRITE);
db.get(`SELECT * FROM users WHERE username = '${username}'`, (err, users) =>{
    if(users === undefined){
        return res.json("Incorrect username or password");
    }
    if(users.username){
      const dbPassword = users.password;
      bcrypt.compare(password, dbPassword).then((result)=>{
        if(result){
          token = jwt.sign({exp: Math.floor(Date.now() / 1000) + (60 * 60), username: username}, config.secret);
          usersLogin.addUserToken(username, token);
          res.json({
            success: true,
            message: 'Authentication successful!',
            token: token,
          });
        }
```

Kuva 17. Kirjautuminen

Kirjautumisen yhteydessä lisätään tämä token users-tietokantaan käyttäjän session -sarakkeeseen, jota verrataan selaimesta tulevaan token:in, kun käyttäjä yrittää lisätä lyhytlinkkejä tietokantaan. Tämä token poistetaan tietokannasta, kun käyttäjä kirjautuu ulos palvelusta.

Lyhytlinkkejä lisäävää funktiota kutsuttaessa tarkistetaan ensimmäisenä, onko token voimassa. Tämä tapahtuu omassa funktiossa, jossa otetaan pyynnöstä irti token, jos sellainen löytyy, jonka jälkeen siitä otetaan alku osa ”Bearer” pois. Jwt-moduuli tarkistaa, onko tämä kyseinen token voimassa. Jos on, lähdetään tarkistamaan, löytyykö vastaava token tietokannasta. (Kuva 18.)

```
async function checkToken(req, res, next){
  let tokenValid = false;
  let token = req.headers['x-access-token'] || req.headers['authorization'] || req.authorization;
  if (token) {
    token = token.slice(7, token.length);
  }
  tokenValid = await checkTokenDatabase(token);
```

Kuva 18. Token:n tarkistaminen

Lyhytlinkkejä lisäävä funktio lisää ja hakee tietoja kutsussa tulevan linkintietojen lisäksi lisääjästä, vanhenemis- sekä alkamisajasta. Se myös tarvittaessa tekee linkille lyhytlinkin, jos sitä ei ole kutsussa määritelty. Tämä tapahtuu erillisessä funktiossa, joka muodostaa aakkosista ja kymmenluvuista muodostetun 4 merkin mittaisen yhdistelmän. (Kuva 19.)

```
async generateRandomString(){
  let length = 4;
  let charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
  let retVal = "";
  let uniq = true;
  for (var i = 0, n = charset.length; i < length; ++i) {
    retVal += charset.charAt(Math.floor(Math.random()  * n));
  }
   uniq = await addingExtras.checkIfUniq(retVal);
```

Kuva 19. Satunnainen merkkijono

Kun merkkijono on luotu, tarkistaa toinen funktio, ettei vastaavaa merkkijonoa jo löydy tietokannasta (kuva 20).

```
let uniq = true;
const db = new sqlite3.Database('db/tietokanta.db');
db.each(`SELECT short_code FROM urls`, (err, urls) => {
  if (err) {
    console.error(err.message);
  }else{
    if(urls.short_code == shortcode ){
      uniq = false;
    }
  }
```

Kuva 20. Yksilöllisyyden tarkistus

Jos vastaavanlainen merkkijono löytyy, kutsuu funktio itseään uudestaan niin kauan, että ainutlaatuinen yhdistelmä syntyy, jonka jälkeen lyhytlinkki lisätään tietokantaan ja selainpäälle palautetaan tieto lyhytlinkistä.

3.5 Automatisoitu järjestelmän hallinta

Nyt kun sovelluksen päätoiminnot on tehty, voidaan sille lähteä tekemään automatisoituja lisätoimintoja. Tähän tarkoitukseen on tehty avuksi Node-schedule-moduuli, jolla voidaan ajaa funktioita tiettyinä kellonaikoina, sekä Nodemailer-moduuli, jolla voidaan lähettää sähköposti-ilmoituksia käyttäjille. Tätä tarkoitusta varten on hyvä tehdä erillinen sähköpostiosoite. Tässä työssä käytettävä sähköpostipalvelu on Googlen Gmail. Kun riippuvuudet on asennettu, voidaan linkkien vanhenemisajan tarkistava ja poistava funktion asettaa käynnistymään päivittäin klo 00.00 (kuva 21).

```
schedule.scheduleJob('0 0 * * *', () => {
  scheduledJob.ScheduledJob();
});
```

Kuva 21. Ajastettu funktio

Jotta sähköpostien lähettäminen onnistuisi tällaisilta ns. tuntemattomilta sovelluksilta, on TLS-rajoitukset poistettava sovelluksen puolella. Sähköpostipalvelusta riippuen täytyy myös usein sallia tuntemattomien sovellusten sähköpostin lähettäminen.

Ajastetun funktion ensimmäinen SQL-kutsu hakee kaikki urls-taulukon solut, joiden expires-sarakkeen unix-ajan arvo on pienempi kuin nykyhetki. Tämä tarkistetaan SQLiten strftime-komennolla, jolla voidaan helposti verrata erimuodoissa olevia aikoja toisiinsa ja muuttaa nykyhetken aika eri aikamuotoihin. Kun nykyhetkeä pienemmällä aika-arvolla oleva solu löytyy, otetaan sen

tiedot talteen objektiin myöhemmin tehtävää ilmoitusta varten. Tarkistetaan, onko tämän vanhenevan linkin omistajalla jo muita vanhenevia linkkejä sisältäviä objekteja. Jos ei, lisätään tämä objekti poistettavien linkkien taulukkoon. Mikäli muita objekteja löytyi, lisätään tähän löydettyyn objektiin uuden objektin tiedot. (Kuva 22.)

```
db.each(`SELECT * FROM urls WHERE strftime(expires) < strftime('%s', 'now')`, (err, urls) => {
  if (err) {
    console.error(err.message);
  }else{
    let userObj = { id : urls.user_id ,url : urls.url, short_code : urls.short_code};
    let uniq = true;
    usersDel.forEach(user => {
      if(user.id == userObj.id){
        uniq = false;
      }
    });
    if(uniq){
      usersDel.push(userObj);
    }else{
      usersDel.forEach(user => {
        if(user.id == userObj.id){
          user.url = user.url + "," + userObj.url;
          user.short_code = user.short_code + "," + userObj.short_code;
        }
      });
    }
  }
```

Kuva 22. Vanhenevien linkkien tarkistus

Kun kaikki vanhenevat linkit ovat löytyneet, käydään läpi poistettavien linkkien -taulukko. Jokaisesta taulukosta löytyvästä objektista otetaan irti käyttäjän id, jota verrata users-taulukon käyttäjien yksilöiviin tunnuksiin. Kun tunnukset vastaavat toisiaan, valmistellaan tälle käyttäjälle sähköposti-ilmoitus poistosta käyttäen Nodemailer-moduulin createTransport-komentoa. Se tarvitsee toimiakseen tiedon siitä, mitä sähköposti palvelua lähettäjä käyttää, sekä lähettäjän sähköpostiosoitteen ja salasanan. Itse viesti lähetetään tämän komennon alikomennolla sendMail. Sille annetaan lähetettävän viestin tiedot, joissa on lähettäjän ja vastaanottajan sähköpostiosoite, aihe sekä viesti. (Kuva 23.)

```
usersDel.forEach(user =>{
  db.each(`SELECT * FROM users WHERE id = '${user.id}'`, (err, urls) => {
    if (err) {
      console.error(err.message);
    }
    else{
      let transporter = nodemailer.createTransport({
        service: 'gmail',
        auth: {
          user: '******@gmail.com',
          pass: '******'
        }
      });
      let mailOptions = {
        from: '******@gmail.com',
        to: urls.email,
        subject: 'Your shortlinks have been deleted.',
        text: `This is automatically sent notification message from shortlink application.
        Do not Answer to this message! \n
        Your shortlinks for Urls: ${user.url} have expired and have been deleted.\n
        This does not require any actions on your part. \n\n -ShortLinkBot *Beeb Buub*`
      };
      transporter.sendMail(mailOptions, function(error, info){
        if (error) {
          console.log(error);
        } else {
          console.log('Email of deleted sent: ' + info.response);
        }
      });
    }
```

Kuva 23. Sähköpostin lähetys

Kun käyttäjille on lähetetty viesti poistosta lähtee funktio poistamaan nämä kyseiset linkit tietokannasta. Kun poisto on suoritettu etsitään poistettavien etsinnän tavalla jäljelle jääneet lyhytlinkit, joiden vanhenemispäivään on alle 30 päivää ja joille ei ole merkitty, että vanhenemisesta olisi lähetetty varoitusilmoitus. Näiden lyhytlinkkien omistajille lähetetään varoitusviesti vanhenemisesta ja päivitetään tietokantaan, että varoitus on lähetetty.

3.6 Linkkien muokkaus ja poisto

Jotta muokkaaminen onnistuu, tarvitaan tieto siitä, minkä lyhytlinkin tietoja muutetaan. Tämä sekä muut mahdolliset muutettavat tiedot saadaan pyynnön rungon kautta. Nämä tiedot sidotaan objektiin, jotta niiden eteenpäin lähettäminen toisille funktioille olisi helpompaa. Objektintiedot lähetetään funktiolle, joka hakee annettua lyhytlinkkiä vastaavat vanhat tiedot. Lisäksi pyynnöstä otetaan irti token, jolla myöhemmin vahvistetaan käyttäjän kirjautumisen voimassaolo, sekä se, että onko tällä käyttäjällä valtuuksia muuttaa tätä linkkiä. (Kuva 24.)

```
async Updatelink(req,res){
  let usertoken = req.headers['x-access-token'] || req.headers['authorization'] || req.authorization;
  if (usertoken) {
      usertoken = usertoken.slice(7, usertoken.length);
  }
  let data = {
    url : req.body.url,
    short_code : req.body.shortcode,
    old_short : req.body.oldcode,
    begins : req.body.begins,
    expires : req.body.expires,
    userId : null
  }
  let newdata = await update.getOldData(data);
```

Kuva 24. Lyhytlinkkien muokkaus

Vanhan tiedon tarkistava funktio etsii tietokannasta taulukon, jonka lyhytlinkki vastaa pyynnön mukana tullutta lyhytlinkkiä. Seuraavaksi funktio käy läpi, jätettiinkö joitakin tietoja tyhjäksi pyynnössä. Jos näin oli, muuttaa funktio palautettavalle objektille kyseiseksi arvoksi vanhan arvon tietokannasta. Lisäksi määritellään objektille id linkin omistajan id:n mukaan. (Kuva 25.)

```
async getOldData(data){
  return new Promise((resolve, reject) => {
    const db = new sqlite3.Database('db/tietokanta.db', sqlite3.OPEN_READWRITE);
    db.each(`SELECT * FROM urls WHERE short_code = '${data.old_short}'`, (err, oldData) =>{
      data.url = data.url.length == 0 ? oldData.url : data.url;
      data.old_short = data.old_short;
      data.short_code = data.short_code.length == 0 ? oldData.short_code : data.short_code;
      data.begins = data.begins.length == 0 ? oldData.begins : data.begins;
      data.expires = data.expires.length == 0 ? oldData.expires : data.expires;
      data.userId = oldData.user_id;
    }, (err,n) => {
      if (err) {
        reject(err);
      }else {
        return resolve(data);
      }
    });
  });
}
```

Kuva 25. Vanhojen tietojen tarkistus

Kun objektin tiedot on muokattu, tarkistetaan seuraavaksi, onko käyttäjällä valtuuksia muokata kyseistä linkkiä. Tämä tapahtuu etsimällä käyttäjä, jonka id vastaa objektin id:tä. Kun käyttäjä löytyy, katsotaan, vastaako tämän käyttäjän aktiivinen token objektin token:ia. Jos tokenit vastaavat toisiaan, voidaan olla varmoja siitä, että kirjautunut linkinmuokkaaja on sama, kuin linkintekijä. (Kuva 26.)

```
async IsCorrectUser(token, userId){
  let correctUser = false;
  return new Promise((resolve, reject) => {
    const db = new sqlite3.Database('db/tietokanta.db', sqlite3.OPEN_READWRITE);
    db.each(`SELECT * FROM users WHERE id = '${userId}'`, (err, user) =>{
      if(user.session == token){
        console.log("Same user")
        correctUser = true;
      }
    }, (err,n) => {
      if (err) {
        reject(err);
      }else {
        db.close();
        return resolve(correctUser);
      }
    });
  });
}
```

Kuva 26. Käyttäjän tarkistus

Tämän jälkeen tarkistetaan aiemmin tehdyllä linkkien yksilöllisyydenvarmista-valla-funktiolla, ettei mahdollinen uusi lyhytlinkki ole jo käytössä ja päivitetään linkin tiedot tietokantaan. Linkkien poistotoiminnallisuus toimii päivitys toiminnallisuutta mukaillen.

3.7 Käyttäjienhallinta

Käyttäjien omia tietoja, sekä heidän tekemiään linkkejä voivat muokata käyttäjä itse sekä manuaalisesti tietokantaan lisätyn hallinnoijan roolin saaneet käyttäjät. Hallinnoijalla on pääsy kaikkiin lyhytlinkkeihin sekä käyttäjiin. Hallinnoijana poisto ja muokkaus toiminnot tapahtuvat samalla tavalla kuin käyttäjän kanssa. Erona kuitenkin on, että tällöin myös tarkistetaan käyttäjän rooli. Roolin on oltava 1, jotta käyttäjä voi suorittaa hallinnoijan tehtäviä (kuva 27).

```
db.each(`SELECT * FROM users WHERE session = '${token}'`, (err, user) =>{
  if(user.session == token){
    if(user.role == 1){
        admin = true;
    }
  }
```

Kuva 27. Hallinnoijan tarkistus

Käyttäjät voivat muuttaa omaa käyttäjänimeään, salasanaansa ja sähköpostiosoitettaan. Kun kirjautunut käyttäjä pyrkii tekemään muutoksia tietoihinsa, otetaan tältä ylös pyynnön mukana tuleva token ja vaaditaan häntä antamaan salasana, jota verrataan tietokannan salasanaan, jonka käyttäjä token vastaa

annettua token:ia (kuva 28). Salasanan ollessa oikein käyttäjä voi muokata tietojaan.

```
db.get(`SELECT * FROM users WHERE session = '${usertoken}'`, (err, users) =>{
  const dbPassword = data.oldpassword;
  bcrypt.compare(dbPassword, users.password).then((result)=>{
    if(result){
      return resolve(true);
    }
    else{
      return resolve(false);
    }
```

Kuva 28. Käyttäjän todennus

Käyttäjillä on oltava myös mahdollisuus saada uusi salasana, jos vanha salasana unohtuu. Kun selainpäästä saadaan salasananpalautuksia käsittelevään polkuun kutsu, jonka mukana on sähköpostiosoite, luodaan käyttäjälle uusi salasana ja lisätään sen salattuversio tietokantaan (kuva 29).

```
async generatePassword(email){
  let password = "";
  let length = 6;
  let charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
  for (var i = 0, n = charset.length; i < length; ++i) {
    password += charset.charAt(Math.floor(Math.random()  * n));
  }
  const saltRounds = 10;
  let databasepassword = await bcrypt.hash(password, saltRounds);
  return new Promise((resolve, reject) => {
    const db = new sqlite3.Database('db/tietokanta.db');
    db.run('PRAGMA busy_timeout = 30000');
    db.configure("busyTimeout", 30000);
    let update = db.prepare(`UPDATE users SET password=? WHERE email = ?`, (err) => {
    }, (err,n) => {
      if (err) {
        reject(err);
      }else {
        update.run(databasepassword, email);
        return resolve(password);
      }
```

Kuva 29. Uuden salasanan luonti

Luotu salaamaton salasana lähetetään käyttäjän sähköpostiin aiemmin tehdyn poistovaroituksen tavalla. Käyttäjä voi nyt kirjautua sisään käyttäen vanhaa tunnustaan ja uutta salasanaansa.

4 PÄÄTÄNTÖ

Nyt on saatu valmiiksi toimiva Node.js backend-sovellus, jolla voidaan tehdä lyhytlinkkejä, käyttää niitä, sekä hallinnoida käyttäjiä, heidän tietojaan, sekä heidän tekemiään lyhytlinkkejä. Sovellukselle voitaisiin seuraavaksi lähteä tekemään selainpäätä, jolla käyttäjät voisivat hoitaa linkkien luonnin, muokkaamisen, poiston ja omat sisäänkirjautumisensa. Tämän voisi toteuttaa käyttäen esimerkiksi Javasriptin React-kirjastoa. Ennen kuin sovellusta lähdettäisiin julkaisemaan, olisi sille hyvä tehdä testauksia useiden eri testaajien toimesta, jotta mahdolliset ongelmat löydettäisiin.

Koin tämän sovelluksen toteuttamisen hyvin opettavaiseksi sekä palkitsevaksi. Alussa sovelluksen tekeminen tuntui hieman haastavalle, sillä monet sovelluksessa olevat tekniikat olivat minulle täysin uusia. Tämä tunne kuitenkin väistyi pian, kun sain ensimmäiset uudet asiat toimimaan. Tämä antoi itsevarmuutta ja sovellus valmistui yllättävän nopeassa tahdissa. Vaikka sovelluksessa onkin paljon erilaisia ominaisuuksia, ovat ne kuitenkin hyvin usein samankaltaisia keskenään ja riitti, että aiemmin tehtyjä funktioita vain muokkasi.

Tämän työn tekeminen opetti minulle, miten paljon erilaisia asioita tulee ottaa huomioon lähdettäessä tekemään tämän mittakaavan sovellusta. Osaan tulevaisuudessa arvioida paremmin, miten paljon aikaa tämän tason sovelluksen tekeminen minulta vaatii ja millaisiin ongelmiin tulen todennäköisesti törmäämään. Työtä tehdessä selkeytyi, kuinka tärkeää koodin helposti luettavana pitäminen on. Käytin välillä melko paljon aikaa mm. funktioiden eri luokkiin jakoon ja yleiseen koodin siistimiseen. Tämä auttoi varsinkin, jos täytyi muokata aikaisemmin tehtyä koodia. Jos tekisin vastaavan työn uudestaan, miettisin jo alusta alkaen, miten tulen jaottelemaan asiat eri luokkiin ja tiedostoihin. Myös paikoin muuttujien nimet saattavat olla joillekin koodia tarkasteleville epäselviä. Aion kiinnittää tähän enemmän huomiota tulevaisuuden projekteissani.

LÄHTEET

Agile Education Research. 2019. Web-palvelinohjelmointi JAVA 2019. WWW-dokumentti. Saatavissa: https://web-palvelinohjelmointi-19.mooc.fi/osa-6/4-rajapinnat-ja-rest [viitattu 10.8.2020].

Codecademy s.a. What is REST. Artikkeli. Saatavissa: https://www.codecademy.com/articles/what-is-rest [viitattu 14.7.2020].

Express s.a.a. Express. WWW-dokumentti. Saatavissa: https://expressjs.com/ [viitattu 1.6.2020].

Express s.a.b. Using Express middleware. WWW-dokumentti. Saatavissa: https://expressjs.com/en/guide/using-middleware.html [viitattu 1.6.2020].

JWT s.a. Introduction to JSON Web Tokens. WWW-dokumentti. Saatavissa: https://jwt.io/introduction/ [viitattu 9.10.2020].

Node.js s.a. About. WWW-dokumentti. Saatavissa: https://nodejs.org/en/about/ [viitattu 18.5.2020].

Npm s.a.a. About npm. WWW-dokumentti. Saatavissa: https://www.npmjs.com/about/ [viitattu 20.7.2020].

Npm s.a.b. JSON web token. WWW-dokumentti. Saatavissa: https://www.npmjs.com/package/jsonwebtoken [viitattu 13.10.2020].

O'Dell, J. 2012. Node.js creator Ryan Dahl steps away from Node's day-to-day. *VentureBeat* 30.1.2012. Verkkolehti. Saatavissa https://venturebeat.com/2012/01/30/dahl-out-mike-drop/ [viitattu 30.5.2020].

Sharma, A. 2017. Story of NodeJs : How JavaScript took over the web. WWW-dokumentti. Saatavissa: https://medium.com/@abhi_/riseofnode-60d8b17c8182 [viitattu 24.5.2020].

SQLite s.a.a. About SQLite. WWW-dokumentti. Saatavissa:

https://www.sqlite.org/about.html/ [viitattu 21.7.2020].

SQLite s.a.b. SQLite is a Self Containded System. WWW-dokumentti. Saatavissa: https://www.sqlite.org/selfcontained.html [viitattu 21.7.2020].

SQLite s.a.c. Most Widely Deployed and Used Database Engine. WWW-dokumentti. Saatavissa: https://www.sqlite.org/mostdeployed.html [viitattu 21.7.2020].

SQLite s.a.d. SQLite is Serverless. WWW-dokumentti. Saatavissa: https://www.sqlite.org/serverless.html [viitattu 21.7.2020].

SQLite tutorial s.a.a. SQLite Create Table. WWW-dokumentti. Saatavissa: https://www.sqlitetutorial.net/sqlite-create-table/ Viitattu [13.8.2020].

SQLite tutorial s.a.b. SQLite Select. WWW-dokumentti. Saatavissa: https://www.sqlitetutorial.net/sqlite-select/ [Viitattu 13.8.2020].

Tsonev, K. 2014. Node.js Blueprints. E-kirja. Birmingham: Packt Publishing. Saatavissa: https://kaakkuri.finna.fi/ [Viitattu 23.5.2020].

Vivah, L. 2017. THE BEGINNER'S GUIDE: Understanding Node.js & Express.js fundamentals. WWW-dokumentti. Saatavissa: https://medium.com/@LindaVivah/the-beginners-guide-understanding-node-js-express-js-fundamentals-e15493462be1 [viitattu 1.6.2020].

W3Schools s.a. SQL DEFAULT Constraint. WWW-dokumentti. Saatavissa: https://www.w3schools.com/sql/sql_default.asp [Viitattu 13.8.2020].

KUVALUETTELO