Opinnäytetyö (AMK)

Tietojenkäsittelyn koulutus

2020

Aleksi Kouhi

VERKKOSOVELLUS YRITYKSEN PROJEKTIEN SEURANTAAN

– Ohjelmistoprojekti Node.js-palvelinympäristössä

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittelyn koulutus

2020 | 41 sivua, 4 liitesivua

Aleksi Kouhi

VERKKOSOVELLUS YRITYKSEN PROJEKTIEN SEURANTAAN

– Ohjelmistoprojekti Node.js-palvelinympäristössä

Opinnäytetyön tarkoituksena oli kehittää toimeksiantajayritykselle verkkosovellus yrityksen urakoiden seurantaan. Sovelluksen avulla pyritään tarjoamaan AKR Oy:lle helppokäyttöinen ja selkeä ympäristö jokaisen meneillään olevan ja suoritetun urakan tarkasteluun.

Työn alussa selvitettiin yrityksen toiveet ja vaatimukset sovelluksen osalta. Sovelluksella haluttiin mahdollistaa yrityksen totetuttamien urakoiden seuranta. Sovellukseen kohdistuvat vaatimukset olivat muun muassa helppokäyttöisyys ja mahdollisuus lisätä, sekä tarkastella urakoita missä vain laitteesta riippumatta.

Raportissa tarkastellaan toimeksiantajan kanssa työhön valittuja teknologioita. Teknologioista kerrotaan lyhyesti niiden tärkeimmät ominaisuudet sekä syyt, miksi ne työhön valittiin. Raportissa käydäään läpi työn toteutus aiemmin valituilla teknologioilla ja kerrotaan miten teknologioita yhdistettiin toimivaksi kokonaisuudeksi. Lopuksi kerrotaan pohdintoja ja ideoita sovelluksen jatkokehitystä varten.

Työ sisältää tärkeimmät vaatimusten mukaiset toiminnallisuudet, kuten uuden urakankohteen luonti ja näiden kohteiden tarkastelu, mutta sovelluksen käyttöliittymä on tämän raportin kirjoitusvaiheesssa vielä tekeillä. Käyttöliittymä ja osa ominaisuuksista vaatii jatkokehitystä yhdessä toimeksiantajayrityksen kanssa, koska tiukan aikataulun vuoksi kaikkia osa-alueita ei saatu valmiiksi oppinnäytetyön aikaikkunan aikana.

ASIASANAT:

Node, Express, MongoDB, Mongoose

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology

2020 | 41 pages, 4 pages in the appendices

Aleksi Kouhi

WEB BASED PROJECT MANAGEMENT SOFTWARE
– Software project in node.js server environment

The purpose of this thesis was to develop a web-based application for the client company to keep track of the company's contracts. The aim of the application is to provide AKR Oy with an easy-to-use environment for reviewing each completed and ongoing contract.

The thesis starts by acquiring the company's requirements for the application. The intended use of the application was to enable the monitoring of completed and ongoing contracts. Requirements of the application included ease of use and the ability to add, as well as view contracts anywhere, regardless of device.

This report reviews the selected technologies for the application. The technologies are briefly described with their main features and the reasons why they were chosen for the application. This report goes through the implementation process of said technologies and explains how the technologies were compined into a functional entity. Finally, reflections and ideas for further development of the application are presented.

The completed work includes the most important functionalities required by the client, such as creating new contracts and reviewing these contracts, but the applications user interface is still in progress at the time of writing this report. The user interface and some of the features require further development together with the client, due to tight schedule.

KEYWORDS:

Node, Express, MongoDB, Mongoose

SISÄLTÖ

LIITTEET

KUVAT

KÄYTETYT LYHENTEET

alustariippumattomuus	Sovellus, joka ei ole sidoksissa tiettyyn käyttöjärjestelmään, Cross-platform
CSS	Koodikieli WWW-sivujen tyylin muokkaamiseen, Cascading Style Sheet
HTML	Koodikieli web-sivujen rakentamiseen, Hypertext Markup Language
JSON	JavaScript Object Notation, tietotyyppi tiedon säilytykseen
käyttöliittymä	Sovelluksen käyttäjälle näkyvä osuus, jolla sovellusta käytetään ja ohjataan.
ohjelmistokehys	Valmis pohja, joka sisältää sovellusten vaatimaa perustoiminnallisuutta; software framework
responsiivinen	Päätelaitteen näytölle mukautuva verkkosivu.
SQL	Structured Query Language, 70-luvulla IBM:n kehittämä relaatiotietokantakyselyihin kehitetty ja 80-luvulla standartoitu kyselykieli, jolla voi hakea, lisätä ja muokata relaatiotietokantoja [1].
Node.js	Avoimen lähekoodin JavaScript-pohjainen palvelinympäristö.
NPM	Node.js Package Manager, Node.js:n pakettien asennustyökalu.
MVC	Arkkitehtuuri, jossa sovellus jakautuu malliin (model), näkymään (view), ja ohjaimeen (controller); model-view-controller

1 JOHDANTO

Mobiililaitteiden käyttö nykypäivän työelämässä on täysin yleistä ja usein jopa pakollista. Hyvin tehdyt yrityksille suunnatut sovellukset helpottavat monen yrityksen arkea ja edesauttavat liiketoiminnan kehitystä. Useat yritykset ovat ottaneet käyttöönsä jonkin projektinseurantasovelluksen, jolla pyritään seuraamaan projektien edistymistä ja kustannuksia.

Vaikka sovellukset helpottavatkin yritysarkea, niillä voi olla monta huonoakin puolta. Käyttöliittymät ovat usein monimutkaisia ja vaikeaselkoisia. Lisäksi sovelluksista voi olla monta eri versiota eri käyttötapauksiin, mutta ne eivät kata silti kaikkea, mitä yritys tarvitsisi.

Tämän opinnäytetyön tarkoituksena on kehittää toimeksiantajayritykselle räätälöity projektinseurantasovellus, joka kattaa yrityksen kaikki tarpeet ja sisältää helppokäyttöisen käyttöliittymän. Sovelluksen avulla pyritään selkeyttämään ja parantamaan yrityksen urakoiden työnkulkua, resurssien seurantaa sekä raportoinnin tekoa. Sovellus on internetpohjainen ja toimii internetselaimella, jolloin sitä on mahdollisuus käyttää missä ja milloin vain alustasta riippumatta. Työ suunniteltiin yrityksen toiveiden mukaisesti, jotta työnkulun ja urakan kokonaistoiminnasta saadaan selkeämpi käsitys yrityksen liiketoiminnan edistämiseksi.

Työssä tarkastellaan aluksi sovelluksen vaatimuksia ja ideoita, joiden pohjalta sovelluksen kehitys aloitetaan. 3. lukuun kootaan teknologiat, joita sovelluksessa ja sen luomisen aikana käytetään. 4. luvussa perehdytään sovelluksen tekoon sekä rakenteeseen. Lopuksi tarkastellaan lopputulosta kokonaisuudessaan, sen sopivuutta yritykselle ja luodaan yhteenveto työstä.

Työn tuloksena syntyi harkitusti suunniteltu ja toimiva projektinhallintasovellus, jolla asiakas saa kirjattua vaivattomasti, mutta turvallisesti urakoidensa tiedot itselleen yhteen paikkaan.

2 TAUSTATIETOJA

Opinnäytetyö tehdään toimeksiantajayritykselle AKR Oy. Yritys toimii rakennusalalla ja hoitaa pääosin kattavasti kattojen huoltoja sekä korjauksia. Yrityksellä ei ole ollut käytössä omaa työnseurantasovellusta tai edes raporttipohjaa erilaisten raporttien teossa. Tähän asti yrityksen raportit urakoista, työtunneista ja kustannuksista on luotu joko käsin tai merkitty yhteistyökumppaneiden järjestelmiin, kun kyseisille kumppaneille on tehty urakoita alihankkijana. Tämän vuoksi yritys tarvitsee oman sovelluksen urakoiden sujuvaan seurantaa sekä raporttien tuottamiseen.

2.1 Ideointi

Alustavasti sovelluksen oli tarkoitus toimia mobiilissa puhelinsovelluksena vain Android-alustalla, mutta mahdollistaa sovelluksen laajentamisen myös IOS-alustalle. Laajennustoiveen vuoksi ensimmäinen idea oli kehittää sovellus Flutter-ohjelmistokehyksen avulla, sillä se mahdollistaa saman koodin rakentamisen sekä Android- että IOS-alustalle ilman merkittäviä muutoksia.

Ongelma pelkän mobiilisovelluksen kehittämisessä tuli ilmi raporttien kirjoittamisen yhteydessä. Täyden raportin kirjoittaminen pelkän puhelimen avulla ei toisi parasta mahdollista käyttäjäkokemusta ja tietokoneen näppäimistön käyttäminen olisi toivottua. Vaihtoehtona olisi erillisen puhelin- ja tietokonesovelluksen luominen. Sovellukset toimisivat erillään, mutta tallentaisivat ja keräisivät tiedot samaan palvelimeen ja näyttäisivät täten samat tiedot. Työstä tulisi kuitenkin mahdollisesti turhan vaikea ylläpitää ja kehitystyö monikertaistuisi.

Työn tilaajan kanssa käydyn pohdinnan tuloksena päädyttiin verkkoselaimeen perustuvaan ratkaisuun, jolloin työnseuranta olisi saatavilla suoraan selaimessa niin tietokoneella kuin puhelimessa alustasta riippumatta. Ideoinnin tuloksena projektinseuranta toteutetaan responsiivisena verkkosovelluksena. Sovellus luodaan node.js palvelinympäristön avulla käyttäen Express-ohjelmistokehystä ja tiedot tallennetaan maksuttomaan MongoDB-tietokantaan.

2.2 Vaatimukset

Vaatimusten kerääminen on olennainen ja tärkeä osa jokaisen ohjelmistoprojektin kehitystä. Ilman tietoa asiakkaan tarpeista kehittäjän on vaikea toteuttaa projektia asiakkaan toiveiden mukaiseksi.

Vaatimusten keräämistä vaikeuttaa usein asiakkaan oma epävarmuus projektin lopputuloksesta. Joustavien tai vaihtelevien vaatimusten kanssa kehitystyö voi hidastua huomattavasti. Ominaisuuksien lisääminen ja poistaminen ohjelmistoprojektista ei aina käy nopeasti ja voi vaatia usean koodirivin uudelleen kirjoittamista. Siksi on tärkeää, että asiakkaalle ja kehittäjälle on selvää, mitä ollaan tekemässä. Tämän työn vaatimukset onnistuttiin keräämään yhden päivän kuluessa asiakastapaamisen yhteydessä. Keräämisen vaikein osuus oli löytää yhteinen kieli asiakkaan kanssa, mutta keskustelun jatkuessa vaatimukset alkoivat muodostua. Vaatimukset sovellukselle ovat seuraavat: helppokäyttöinen käyttöliittymä, mahdollisuus luoda uusia urakkakohteita ja kuntoraportteja missä vain, urakkakohteen kuvaaminen, urakan työtuntien seuraaminen ja kustannusten selkeä erittely.

Tärkeimmät sovellukseen kohdistuvat vaatimukset ovat helppokäyttöinen käyttöliittymä ja mahdollisuus urakkakohteiden tarkasteluun ja luontiin missä vain. Vaatimuksista vaikein toteuttaa on mahdollisesti helppokäyttöinen käyttöliittymä. Muiden vaatimusten visualisoinnin yhteydessä pyritään muodostamaan käyttöliittymää. Käyttöliittymään tehdään tarvittaessa muutoksia, mikäli asiakas kokee sen tarvitsevan yksinkertaistamista.

Vaatimukset on eritelty tarkemmin luvuissa 2.3 ja 2.4 käyttäjätarinoina, joista osa on havainnollistettu käyttötapausten ja käyttötapauskaavioiden avulla.

2.3 Käyttäjätarinat

Vaatimukset kuvataan usein käyttäjätarinoina. Käyttäjätarinat ovat lyhyitä kuvauksia sovelluksen halutusta toiminnasta. Niiden avulla pyritään selkeyttämään sovelluksen toimintaa sen suunnitteluvaiheessa yksinkertaisella ja helposti ymmärrettävällä lauseella, esim. *Uutena sovelluksen käyttäjänä haluan mahdollisuuden kirjautua sovellukseen omilla käyttäjätunnuksillani.*

Opinnäytetyössä tehtävän sovelluksen käyttäjätarinoihin kuuluvat seuraavat:

1. *Käyttäjänä haluan kirjoittaa sovelluksen avulla urakkakohteesta kuntoarvion.*

---

2. *Käyttäjänä haluan mahdollisuuden luoda uusia urakkakohteita.*

---

3. *Käyttäjänä haluan mahdollisuuden tallentaa valmiit urakat sekä raportit myöhempää tarkastelua varten.*

---

4. *Käyttäjänä haluan ottaa ja tallentaa kuvia kohteesta.*

---

5. *Käyttäjänä haluan tarkastella urakkakohteen aikataulua sekä seuraamaan sen etenemistä.*

---

6. *Käyttäjänä haluan pystyä seuraamaan urakkaan käytettävä materiaaleja, niiden määriä sekä kustannuksia.*

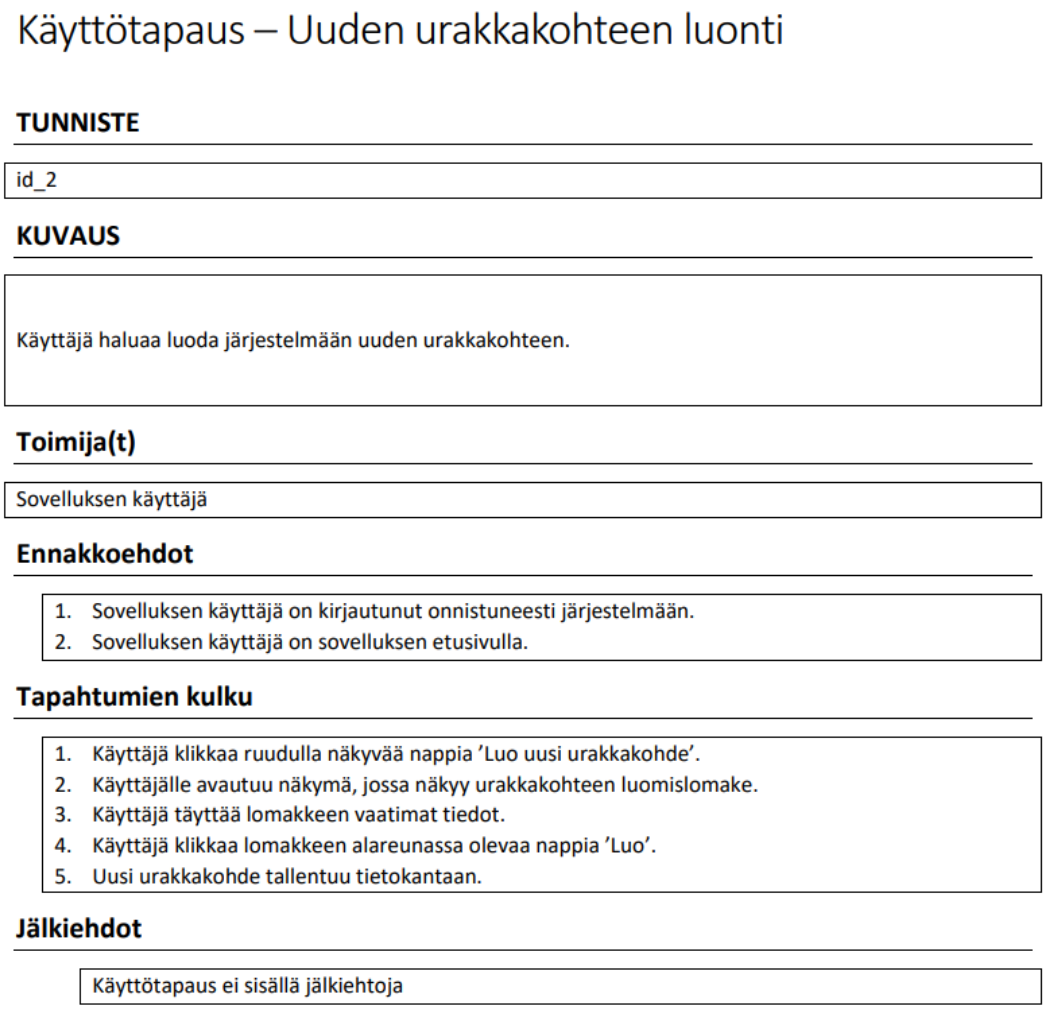
2.4 Käyttötapaukset

Käyttäjätarinoiden hahmottamiseen voidaan luoda käyttötapauksia. Käyttötapauksilla pyritään dokumentoimaan ohjelmiston toimintaa ja auttamaan kehittäjiä ohjelmiston suunnittelussa. Niillä kuvataan käyttäjätarinoiden haluttua toimintaa yksityiskohtaisesti.

Käyttötapauksissa ohjelmiston toivottu toiminta käytiin läpi vaihe vaiheelta, jotta käyttäjän ja ohjelmiston vuorovaikutuksen hahmottaminen helpottuisi. Käyttötapauksille ei ole standardoitua kaavaa, minkä mukaan ne pitäisi kirjoittaa, mutta niistä on hyvä käydä ilmi, mitä halutaan tehdä, kuka haluaa tehdä ja miten toiminnan suorittaminen toteutetaan, sekä mahdolliset tarvittavat esi- ja jälkiehdot [8]. (Kuva 1.)

Työhön luotiin vaatimusten perusteella neljä käyttötapausta, joita hyödynnettiin sovelluksen teossa. Ensimmäinen käyttötapaus kertoo käyttäjältä vaaditut toiminnat uuden kuntoarvion luomiseen ja toisessa käyttötapauksessa käydään läpi käyttäjältä vaaditut toiminnat uuden urakkakohteen luomisessa. Kolmas käyttötapaus kuvaa miten käyttäjä

saa esille tallennetun urakkakohteen tiedot. Neljäs käyttötapaus selittää käyttäjän toimintoja uuden kuvan lisäämisessä urakkakohteesta. Jokainen työhön tehty käyttötapaus on lisätty liitteeksi tähän työhön.



Käyttötapaus – Uuden urakkakohteen luonti

TUNNISTE

id_2

KUVAUS

Käyttäjä haluaa luoda järjestelmään uuden urakkakohteen.

Toimija(t)

Sovelluksen käyttäjä

Ennakkoehdot

1. Sovelluksen käyttäjä on kirjautunut onnistuneesti järjestelmään.
2. Sovelluksen käyttäjä on sovelluksen etusivulla.

Tapahtumien kulku

1. Käyttäjä klikkaa ruudulla näkyvää nappia 'Luo uusi urakkakohde'.
2. Käyttäjälle avautuu näkymä, jossa näkyy urakkakohteen luomislomake.
3. Käyttäjä täyttää lomakkeen vaatimat tiedot.
4. Käyttäjä klikkaa lomakkeen alareunassa olevaa nappia 'Luo'.
5. Uusi urakkakohde tallentuu tietokantaan.

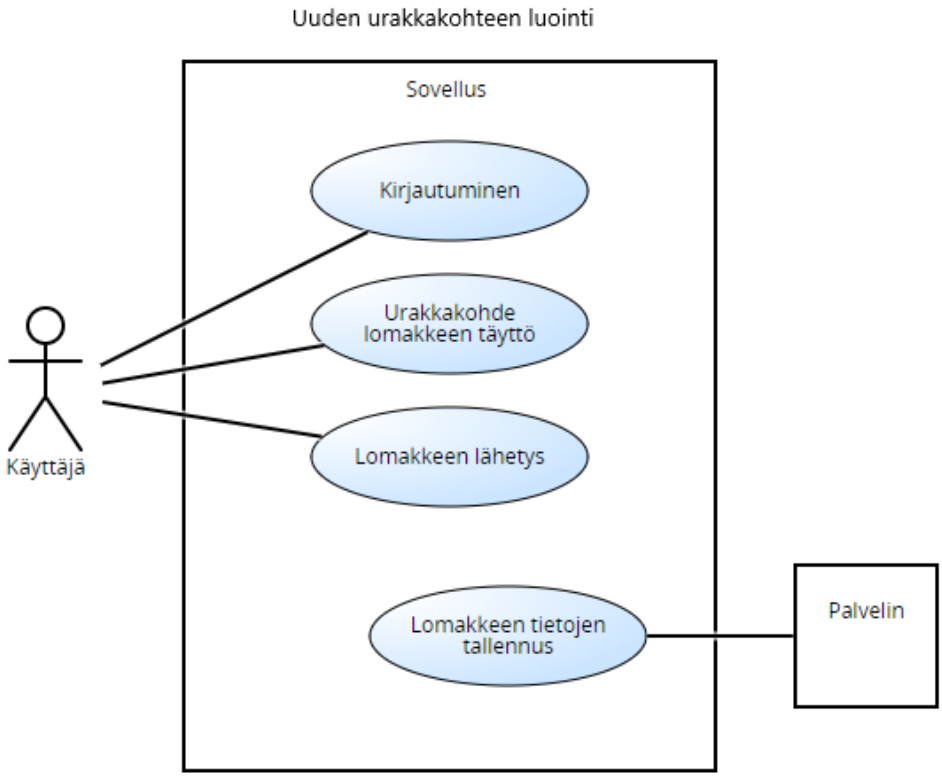
Jälkiehdot

Käyttötapaus ei sisällä jälkiehtoja

Kuva 1. Käyttötapaus uuden urakkakohteen luonnista. Tapaus kuvaa koko urakkakohteen luontitapahtuman ja kuvaa käyttöliittymää.

Käyttötapauksia kuvaamaan liitetään usein mukaan käyttötapauskaavio, jolla käyttötapausta kuvataan visuaalisesti. Käyttötapauskaaviot ovat usein yksinkertaisia, eikä niiden tarkoitus ole kertoa tarkasti koko tapauksen toimintaa, mutta kaavioista käy ilmi toimijoiden väliset suhteet ja toiminnat. Nopeiden piirrosten tekeminen on hyvä tapa hahmottaa käyttötapauksen toiminta ja samalla suunnitella käyttöliittymää. Työssä luodut

käyttötapaukset kertovat ja kuvaavat hyvin myös sovelluksen käyttöliittymää, joten suurin osa käyttöliittymästä toteutetaan käyttötapausten pohjalta. (Kuva 2.)



Kuva 2. Käyttötapauskaavio uuden urakkakohteen luonnista

.

3 KÄYTETYT TEKNOLOGIAT

Tässä luvuissa eritellään työn suunnittelussa ja itse työssä käytetyt teknologiat.

3.1 GitHub

GitHub on verkkosivu, joka mahdollistaa Git-versionhallintaa hyödyntävien sovellusprojektien pilvisäilytyksen. Sen kehitys sai alkunsa Yhdysvalloissa vuonna 2007 ja julkaistiin vuonna 2008. Git on ohjelmistoympäristö, jota käytetään pääasiassa komentorivin avulla. Vaikka se luotiin komentoriviohjelmaksi, on tarjolla myös graafinen käyttöliittymä, mikä helpottaa kokemattomimpia käyttäjiä Gitin toiminnassa. Graafinen käyttöliittymä auttaa aloittelijoita Gitin käytössä ja siihen tutustumisessa.

Ohjelmakoodin pilvisäilytyksen lisäksi GitHub tarjoaa myös virheenseurantajärjestelmän, tilastotietoja ja tehtävienhallintaa [2]. Githubiin voi luoda tilin kuka vain täysin ilmaiseksi, mutta ilmaistileille on tehty pieniä rajoituksia koodin säilytyksen ja ylläpidon suhteen. Täydet oikeudet haluava kehitystiimi maksaa 4 euroa kuukaudessa ja yrityksille samojen oikeuksien lähtöhinta on 21 euroa kuukaudessa [3][4].

3.2 Node.js

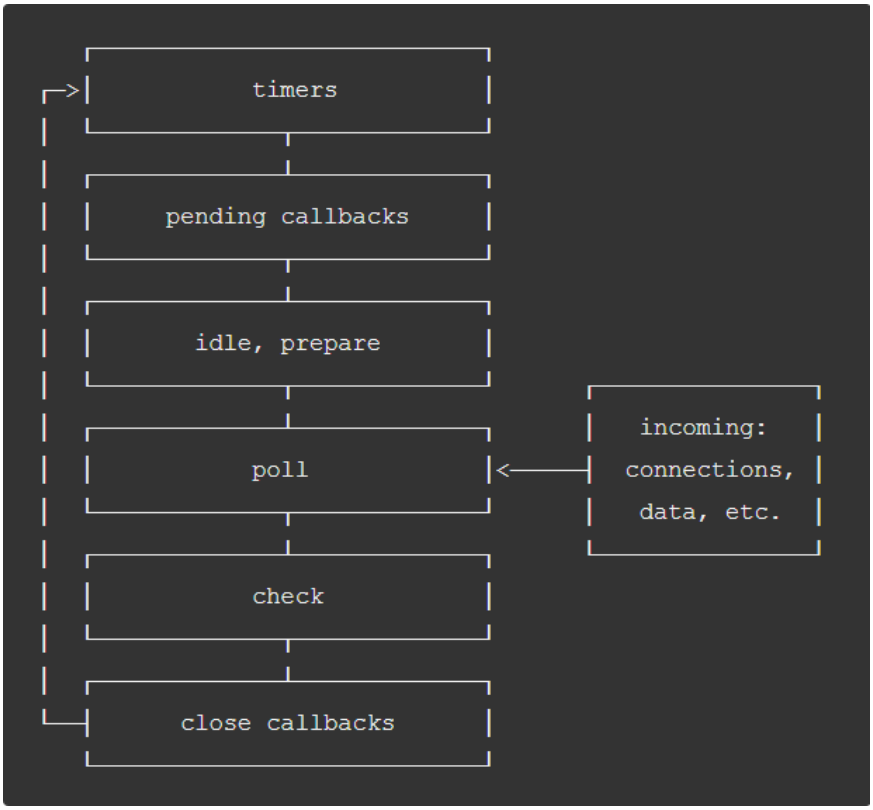
Node.js on Ryan Dahlin vuonna 2009 kehittämä JavaScriptillä toimiva avoimen lähdekoodin asynkroninen tapahtumapohjainen palvelin. Se sallii JavaScript-ohjelmakoodin ajon selaimen ulkopuolella ja perustuu Googlen Chrome V8-moottoriin [4] [5, s. 10]. Node.js siis mahdollistaa JavaScriptin käytön pätevänä palvelinpuolen ohjelmointikielenä, jonka ansiosta koko web-sovelluksen logiikka voidaan rakentaa yhden kielen avulla.

Kun Node-ohjelma käynnistyy, kaikki sovelluksen sisältämä koodi ladataan muistiin. Kaikkea koodia ei kuitenkaan ajeta saman tien, vaan tarvittava koodi suoritetaan, kun koodia tarvitseva tapahtuma kutsuu sitä [5, s. 10].

3.2.1 Tapahtumasilmukka

Node.js on yksisäikeinen ohjelma eli se käyttää pääasiassa vain yhtä säiettä. Säie vastaa ohjelmankoodin suorittamisesta ja perinteisesti yksi säie aloittaa ja suorittaa aina yhden tehtävän tai toiminnon ohjelmassa, mitä enemmän tehtäviä ohjelmassa suoritetaan samanaikaisesti, sitä enemmän säikeitä pitää olla käytössä. Node.js kuitenkin pystyy suorittamaan useita toimintoja samassa säikeessä tapahtuma silmukan ansiosta. Useampia säikeitä tarvitaan, jos toimintoa ei jostain syystä voida suorittaa pääsäikeessä. Tapahtumasilmukan ansiosta node.js pystyy suorittamaan useita tehtäviä nopeasti ja tehokkaasti vain yhdellä säikeellä. [5, s. 20].

Event Loop eli tapahtumasilmukka aloittaa toimintansa heti ohjelman alettua ja lopettaa toiminnan vasta kun ohjelman suoritus lopetetaan. Node.js:n tapahtumasilmukka sallii sen asynkronisen ja ei-sulkevan toiminnot. Ei-sulkevalla eli non-blocking-ominaisuudella tarkoitetaan Node.js:n kykyä suorittaa koodia taustalla takaisinkutsujen avulla [7]. (Kuva 3.)



Kuva 3. Node.js tapahtumasilmukan toimintaa kuvaava kuva [7].

3.2.2 NPM

Node Package Manager (NPM) on Node.js projekteille luotu verkossa toimiva paketin hallinta-, ylläpito- ja asennustyökalu. NPM:n avulla Node-projekteihin voi asentaa, ladata sekä hallita 3. osapuolen lisäosia NPM-kirjastosta [9]. NPM-kirjasto sisältää suuren määrän avoimenlähdekoodin paketteja, joita voi vapaasti liittää osaksi omaa Node-projektia. Pakettien asennuksessa on hyvä käyttää liitettä '--save' asennuskomennon yhteydessä. Tällä liitteellä varmistetaan, että asennettava paketti tallennetaan ja merkitään riippuvuudeksi sovelluksen packages.json-tiedostoon. Jokainen node-projekti tarvitsee packages.json tiedoston. Tähän tiedostoon merkitään kaikki sovelluksen toimintaan tarvittavat paketit. Packages.json-tiedosto luodaan automaattisesti Node-projektin yhteydessä [5, s .47]. Tässä työssä NPM:n käyttö on keskeistä, sillä työssä hyödynnetään useita lisäosia, jotka helpottavat sovelluksen kehitystyötä.

3.3 Express.js

Ohjelmistokehykset nopeuttavat kehitystyötä tarjoamalla valmiin pohjan, jonka päälle ohjelmat tuotetaan. Nodelle on saatavia useita eri ohjelmistokehyksiä, mutta tässä työssä käytetään Node.js-ympäristön suosituinta Express.js ohjelmistokehystä. Express.js julkaistiin vuonna 2010 ja se on yksi noden vanhimmista ja suosituimmista kehyksistä. Sen käyttö ohjelmistokehyksenä kasvaa joka vuosi. Se on kevyt ja sisältää paljon valmiiksi luotuja perustoiminnallisuuteen tarvittavia ominaisuuksia ja työkaluja, jotka helpottavat ja nopeuttavat Node.js applikaatioiden kehitystä. Express.js helpottaa muun muassa Node.js-ohjelman reititystä ja middlewaren käyttöä, kuten tässä työssä tullaan huomaamaan [11, s. 6].

3.4 MongoDB ja Mongoose

MongoDB on NoSQL-tietokanta, jossa tiedot tallennetaan JSON-objekteina [6]. JSON-objektien käyttö JavaScript toiminta ympäristössä on luonnollista, sillä samaa tietomuotoa käytetään javascriptin kanssa paljon. JSON:in toiminta on kuitenkin JavaScriptistä riippumatonta, mutta on silti selkeä tapa esittää tietoa.

Tähän työhön valitsin MongoDB-tietokannan, koska siitä on minulla aikaisempaa kokemusta ja sen käyttö on maksutonta.

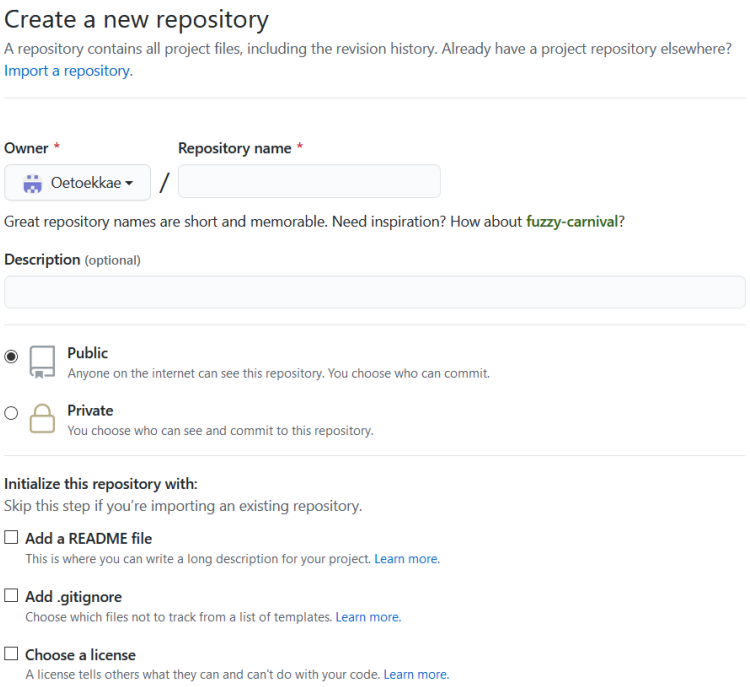
Mongoose on nodessa käytettävä kirjasto, joka helpottaa MongoDB tietokannan ylläpitoa sekä järjestystä [5, s. 165].

4 TOTEUTUS

Tässä luvussa käydään läpi sovelluksen toteutus aiemmin mainituilla teknologioilla.

4.1 Git ja projektin alustus

Ensimmäinen asia tämän ohjelmistoprojektin toteutuksessa on luoda projektille oma git-ympäristö. Gitin pystyttäminen on vaivaton ja nopea prosessi. Git-ympäristön voi luoda suoraan komentolinjalta tai github.com osoitteessa. Tämän projektin git-ympäristö luotiin kokonaan nettisivujen kautta. Kuvassa 4 näkyy nettisivujen kautta täytettävä luomislomake.



Kuva 4. Github.com-sivuston repositorion luomislomake.

Sivuston kautta repositorion luonti omalle projektille on nopeaa. Projektiin sisällytetään ‘readme’ ja ‘gitingnore’ tiedostot. Näistä ensimmäiseen voidaan kirjoittaa kuvaus projektista sekä tärkeää tietoa siitä ja sen käytöstä. Jälkimmäinen tiedosto kertoo gitille, mitä tiedostoja tai tiedostotyyppejä ei gitin tarvitse seurata. Tässä projektissa hyödynnettiin gitin valmista node.js:lle luotua gitignore-tiedostoa, joka sisältää kaikki node.js:ään

yhdistettävät tiedostotyypit, joita gitin ei tarvitse seurata. Tiedostoon voi halutessaan lisätä muitakin tiedostotyyppejä manuaalisesti myöhemmin. (Kuva 5.)

Kuva 5. Pieni osa gitignore-tiedostosta.

Repositorion luomisen jälkeen projektin voi kopioida omalle tietokoneelle käyttämällä joko https- tai ssh-yhteyttä.

Tämän projektin kopiointi toteutettiin https-yhteyden avulla ajamalla komento ‘`git clone [projektin https osoite]`’ komentoriviin. (Kuva 6.)

Kuva 6. Projektin kopiointi gitin avulla.

Gitin pystyttämisen jälkeen on aika alustaa node-projekti. Projektin alustaminen aloitetaan ajamalla komentolinjassa komento ‘`npm init`’. Tämän komennon kautta saadaan esille lomake johon täytetään projektin tärkeimmät tiedot, kuten nimi, kuvaus, versio, git repositorio, tekijä yms. Tiedot tallentuvat package.json tiedostoon. Tiedostossa käy myös ilmi kaikki projektin ajamiseen tarvittavat myöhemmin erikseen asennettavat paketit sekä ohjelmointikehys.

4.1.1 Kansiorakenne

Seuraava askel oli luoda projektiin selkeä kansiorakenne. Juurikansio sisältää projektin kannalta välttämättömimmät tiedostot kuten index.js, jolla koko projekti ja node-palvelin käynnistetään. Index.js sisältää myös palvelinyhteyden määrittelyn sekä sovelluksen alkumäärittelyitä. Projekti noudattaa muuten MVC-arkkitehtuuria, jossa sovelluksessa käytettävät tietomallit, näkymät ja kontrollerit ovat kaikki omissa kansioissaan ‘models’, ‘views’ ja ‘controllers’. Näin projektin rakenne pysyy yksinkertaisena ja helposti navigoitavissa.

Models

Models-kansiossa sijaitsee jokainen sovelluksessa käytettävä tietoamalli. Nämä mallit määrittelevät tietokantaan tallennettavan tiedon muodon. Tässä työssä hyödynnetään kolmea eri mallia: käyttäjät, urakat ja kuntoarviot.

Tietokannan malleista puhutaan tarkemmin tietokantaan keskittyvän kappaleen luvussa 4.4.1.

Views

Näkymät-kansio sisältää itse käyttäjälle näkyvät html-tiedostot. Tiedostot ovat kansiossa ejs-sivupohjina, josta ne muutetaan selaimelle sopivaksi html-muotoon (ks. luku 4.2). Kansio on jaettu alakansioihin eri aihe alueiden kesken. Esimerkiksi kaikki urakoihin liittyvät html-tiedostot ovat omassa alakansiossa “cases”.

Controllers

Controllers-kansio sisältävää sovelluksen palvelimen logiikan. Saman aihealueen toiminnot sisällytetään yhteen ohjaintiedostoon helpottamaan sovelluksen ylläpitoa, esimerkiksi jokainen käyttäjän ja sovelluksen välinen kommunikointi, kuten sisään- ja uloskirjautuminen ovat sisällytetty samaan userCTRL.js-tiedostoon. Samoin urakoiden luonti, tarkastelu ja muokkauspyynnöt ovat sisällytetty urakoille tarkoitettuun

caseCRTL.js ohjaintiedostoon. Ohjaintiedostojen sisältämä logiikka noudattaa CRUD-mallia (ks. luku 4.4.2).

4.1.2 Palvelimen käynnistys

Palvelimen käynnistäminen ei vaadi montaa riviä koodia toimiakseen. Nodelle täytyy ensin kertoa, mitä paketteja koodissa halutaan hyödyntää vaatimalla ne tiedostoon `require()`-funktiolla. Palvelimen konfiguroinnissa arvojen asetus tapahtuu set() ja use()-funktioiden avulla. Työssä käytetään express-kehystä, joten node-pitää konfiguroida myös hyödyntämään sitä. Express-moduuli saadaan käyttöön liittämällä siitä instanssi vakiomuuttujaan index.js-tiedoston koodissa. Express moduulin palvelin sijoitetaan omaan vakiomuuttujaan '`app`'. App muuttujan avulla päästään käsiksi suurimpaan osaan expressin tarjoamista ominaisuuksista. Express moduulin käyttöönoton lisäksi index.js-tiedostossa määritellään palvelimen tiedot ja reitityksen ohjaus routes.js-tiedostoon, mikä käsittelee pyynnöt ja laukaisee ohjaimissa sijaitsevat middleware-funktiot. Palvelin määritellään myös käyttämään HTML-tiedostojen esittämiseen ejs(Embedded JavaScript Templates)-lisäosaa, sekä käsittelemään pyynnöissä tulevaa dataa `urlencoded()`-funktiolla.

Lopuksi palvelin käynnistetään `listen()`-funktion avulla. Pyyntöjen käsittelyssä jokainen pyyntö tulostetaan konsoliin kehitystyön helpottamiseksi. Kuvassa 7 havainnollistetaan index.js-tiedoston sisältämä koodi, jolla palvelin käynnistetään.

```
"use strict"

//require packages, router, other files
const express = require("express");
const layouts = require("express-ejs-layouts");
const router = require("./router");
//get instance of express
const app = express();

//set app port, templating engine,
//location of static html and body-pareser for data processing
app.set("port", process.env.PORT || 3000);
app.set("view engine", "ejs");
app.use(layouts);
app.use(express.static("public"));
app.use(
    express.urlencoded({
        extended: false
    })
);

// Routes
app.use('/', router);

//console log request
app.use((req, res, next) => {
    console.log(`request made to: ${req.url}`);
    next();
});

app.listen(app.get("port"), () => {
    console.log(`Server running at http://localhost:${app.get("port")}`);
});
```

Kuva 7. Index.js-tiedoston alkumäärittelyt.

4.1.3 Reititys

Reitityksellä tarkoitetaan sisäänkirjautumissivulta johtavien muiden näkymien polkujen ja oheistoimintojen määrittelyä. Esimerkiksi sisäänkirjautumisen yhteydessä annetut tunnukset tarkistetaan ja käyttäjä ohjataan eteenpäin sovellukseen tai takaisin etusivulle, riippuen tunnistautumisen onnistumisesta. Reitityksen työssä hoitaa routes.js -tiedosto, mikä ohjaa pyynnöt polun perusteella oikealle ohjaimelle. Työn reitityksessä käytettävät ohjaimet sijaitsevat omassa alakansiossa (ks. luku 4.1.1) ja toiminnot ovat jaettu usean eri tiedoston välille.

Router.js-tiedostossa kutsutaan Express.js:n sisäistä funktiota `router()`, mistä saatu instanssi tallennetaan `router` vakiomuuttujaan. Index.js-tiedostossa otetut pyynnöt lähetetään suoraan router.js-tiedostoon, jossa niiden sisältämän polun perusteella suoritetaan toimintoja oikeista ohjain tiedostoista takaisinkutsufunktioina. Kun käyttäjä

saapuu sovellukseen, ohjataan hänet kirjautumissivulle. Get()-funktiolla otetaan vastaan GET-metodilla lähetetyt pyynnöt. Router.js tiedostossa get()-funktiossa kutsutaan kotiohjaimen showLogin()-funktiota, mikä siirtää käyttäjän sovelluksen etusivulle. (Kuva 8.)

```
//express and router
const express = require("express");
const router = express.Router();
//controllers
const homeCTRL = require("./controllers/homeCtrl");
const userCTRL = require("./controllers/userCtrl");
//routes
router.get("/", homeCTRL.showLogin);
router.post("/users/login", userCTRL.authenticate);
```

Kuva 8. Router.js-tiedoston polunohjaus etusivulle ja kirjautumisessa.

Käyttäjän sisäänkirjautumisessa palvelimelle lähetetään POST-metodin kautta sisäänkirjautumiseen vaadittavat tiedot “/users/login” polkuun. Router.js-tiedostossa pyyntö otetaan vastaan ja suoritetaan käyttäjäohjaimen sisältä authenticate()-funktio. Funktio varmistaa tiedot Passport.js:n avulla ja ohjaa käyttäjän, joko sovelluksen sisään tai takaisin kirjautumissivulle, riippuen kirjautumisen onnistumisesta. (Kuva 9.)

```
authenticate: passport.authenticate("local", {
    failureRedirect: "/",
    failureFlash: "Sisäänkirjautuminen epäonnistui.",
    successRedirect: "/home",
    successFlash: "Sisäänkirjautuminen onnistui!"
}),
```

Kuva 9. Käyttäjäohjain-tiedoston sisältämä authenticate()-funktio.

4.2 Näkymät

Näkymät ovat sovelluksessa loppukäyttäjälle näkyvät sivut. Työssä käytetään html-sivujen esitykseen ejs(Embedded JavaScript Templates)-lisäosaa. Se mahdollistaa html-sivujen esittämisen ejs-sivupohjien avulla. Eli käytännössä esitetään staattisia html-sivuja, joiden sisältöä voidaan muokata dynaamisesti muualla sovelluksessa, esimerkiksi sisään kirjautuneen käyttäjän perusteella. Kaikki sovelluksen html sisältö löytyy ‘views’

alakansiosta. Jokaisella näkymällä on oma ejs-tiedosto. Sivupohjien perustaksi luodaan layouts.ejs-tiedosto. Tämä tiedosto määrittää eri sivupohjissa olevien elementtien järjestyksen sovelluksessa. Näkymien suunnittelua edesauttoi vaatimusten perusteella tehdyt käyttäjätarinat, joissa kerrotaan tarkasti käyttäjän suorittamat toiminnot näkymissä. (Kuva 10.)

```
<!DOCTYPE html>
<html>

    <head>
        <%- include('partials/head') %>
    </head>

    <body>
        <div class="header">
            <%- include('partials/header') %>
        </div>
        <div class="container">
            <%- body %>
        </div>
        <div class="footer">
            <%- include('partials/footer') %>
        </div>
    </body>

</html>
```

Kuva 10. Layout.ejs, johon liittetty html-sivupohjat.

4.2.1 Sisäänkirjautuminen

Ensimmäinen näkymä on sovelluksen sisäänkirjautumissivu, toisin sanoen index.ejs tiedosto. Tässä näkyy käyttäjälle sisäänkirjautuminen ja koko sovelluksen ilme. Ensin näkymä kannatti suunnitella ja tämän toteutin Photoshop –kuvanmuokkaus ohjelman avulla. Etusivu on ohjelmistoprojektin ainoa näkymä, jonka ulkoasun suunnitteluun käytettiin muutakin kuin kynää ja paperia. Kuva 11 näyttää etusivun alkuperäisen luonnoksen, jonka pohjalta index.ejs sivu tehtiin. (Kuva 12.)

Kuva 11. alustava suunnitelma sisäänkirjautumissivun mobiilinäkymästä.

Kuva 12. Sisäänkirjautumisen lopullinen ilme, mobiilinäkymässä.

Näkymässä on yrityksen logon alla lyhyt tervehdys tekstin ja sisäänkirjautumislomake. Näkymä pyrkii olemaan mahdollisimman helppokäyttöinen ja suoraviivainen. Sivun tyyli on toteutettu kokonaan CSS:llä ja värimaailma pyrkii noudattamaan logon värejä. Tulevat näkymät luodaan noudattaen kirjautumissivun tyyliä.

Sisäänkirjautumis-lomakkeeseen kirjoitetaan käyttäjätunnus sekä salasana. Sisäänkirjautumis-nappula lähettää palvelimelle POST-pyynnön, joka sisältää sekä käyttäjätunnuksen että salasanan. Mikäli tunnukset ovat oikein ja tietokannasta löytyy oikea käyttäjä, jatkaa sovellus eteenpäin itse sovellukseen. Muutoin käyttäjälle näkyy virheilmoitus epäonnistuneesta kirjautumisyrityksestä ja vaaditaan tunnusten uudelleen syöttämistä.

Käyttäjän tietojen turvaaminen

Tietoturva on tietysti tärkeä osa jokaista sovellusta. Tietovuoto on yksi pahimmista asioista, mitä sovelluksen tiedoille voi käydä.

Sovelluksen kirjautumissivulla hoidetaan käyttäjän tunnistaminen. Se on ainoa sovelluksen osa, joka näkyy tunnistautumattomalle käyttäjälle. Käyttäjälle käyttäjätiedot sovellukseen luo järjestelmän ylläpitäjä. Tiedot tallennetaan tietokantaan salattuina Passport.js-väliohjelman avulla. Passport hoitaa käyttäjätunnusten salauksen ja tunnistamisen tehokkaasti sovelluksen taka-alalla. Se myös hyödyntää sessioita ja evästeitä, joiden avulla varmennetaan käyttäjän oikeus navigoida sovelluksessa, sekä kirjataan käyttäjä tarvittaessa ulos, mikäli käyttäjä on ollut pitkään käyttämättä sovellusta. Väliohjelman asennus onnistuu NPM-työkalun avulla syöttämällä komentoriviin komento: `'npm install passport –save'.`

Passport otetaan käyttöön sovelluksen reitityksen yhteydessä router.js-tiedostossa. Palvelin määritellään käyttämään Passport.js:ää sen sisäänrakennetun `initialize()`-funktion avulla ja `session()`-funktiolla kerrotaan palvelinta hyödyntämään Passport.js:n sessioita.

Palvelimen määrityksen jälkeen suoritetaan Passport.js:n määritys. Ensin määritellään strategia, jonka mukaan käyttäjä autentikoidaan. Strategioiden avulla sovellus voidaan määritellä käyttämään useita eri kirjautumisvaihtoehtoja, kuten sisäänkirjautuminen Facebook- tai Google-tilien avulla, mutta tässä työssä käytetään vakiostrategiaa, jossa käytetään käyttäjätunnusta ja salasanaa. Tämä vakiostrategia saadaan “passport-local-mongoose” -paketin kautta. Paketti lisätään käyttäjä-tietomalliin (ks. luku, 4.4.1).

Lopuksi käyttäjät pitää serialisoida ja deserialisoida, jotta Passport.js:n sessiot saadaan käyttöön. Tällä tarkoitetaan käyttäjän datan muuttamista merkkijonoksi evästeisiin ja takaisin alkuperäiseen muotoon. (Kuva 13.)

```
router.use(passport.initialize());
router.use(passport.session());
passport.use(User.createStrategy());
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

Kuva 13. Passport.js-väliohjelman käyttöönotto router.js-tiedostossa.

4.2.2 Etusivu

Sisäänkirjautumisen jälkeen käyttäjälle avautuu näkymä sovelluksen sisäisestä etusivusta. Sivulla käyttäjällä on mahdollista navigoida sovelluksen sisällä käyttäen koko ajan esillä olevaa navigointipalkkia sivun yläosassa. Etusivulla on myös kolme nappulaa, joiden avulla käyttäjä voi luoda uuden urakkakohteen, kuntoarvioraportin tai tarkastella jo luotuja urakkakohteita.

4.2.3 Urakoiden ja kuntoarvioiden luonti

Työn tärkein osuus asiakkaan näkökulmasta on uusien urakoiden lisäys ja näiden tarkastelu sovelluksesta. Tässä ja seuraavassa luvussa käsitellään projektin näitä osa-alueita.

Urakoiden ja kuntoarvioiden luontilomakkeet saa esille heti sovelluksen etusivulta. Sivulla käyttäjälle näkyy lomake, johon pitää täyttää urakkakohteen tiedot ennen kuin kohde voidaan luoda ja tallentaa tietokantaa. (Liite 2.) Lomake vaatii urakan nimen, osoitteen, aloitus ja arvioidun valmistuspäivän sekä kohteen sopimushinnan. Lomakkeeseen on myös mahdollista täyttää kaupungin nimi sekä sijainnin postinumero. Nämä tiedot ovat kuitenkin valinnaisia, sillä urakan sijainti tarkastelu näkymässä haetaan osoitteen perusteella (ks. luku 4.2.4).

Lomakkeeseen täytettävät tiedot tarkastetaan ennen lähetystä HTML:n sisäänrakennetun validoinnin avulla. Validointi varmistaa, että tiedot ovat oikeassa muodossa. Esimerkiksi numeroarvot pitää syöttää numeroina ja vaaditut kentät pitää olla täytettynä ennen kuin lomake voidaan lähettää. Lomakkeen tiedot lähetetään palvelimelle POST-metodilla, jotta palvelin tietää pyynnön sisältävän tietokantaan tallennettavaa tietoa. Kuva 14 näyttää lomakkeen HTML-koodin.

```
<form action="/cases/create" method="POST">
    <div class="case-name-container">
    <label class="case-name-label" for="inputName"><span>Urakan nimi:</span></label>
    <input class="form-control" type="text" name="name" id="inputName" placeholder="Nimi" required>
    </div>
    <label class="case-label" for="jobDescription"><span>Urakan kuvaus:</span></label>
    <textarea class="form-control" rows="4" cols="50" name="description" id="jobDescription" placeholder="kirjoita tähän lyhty kuvaus urakasta..." required></textarea>

    <label class="case-loc-label" for="inputAddress"><span>Osoite:</span></label>
    <input class="form-control" type="text" name="address" id="inputAddress" placeholder="Osoite, esim 'Esimerkkitie 5'" required>

    <label for="inputCity"><span>Kaupunki</span></label>
    <input class="form-control" type="text" name="city" id="inputCity" placeholder="Kaupunki, jossa urakka sijaitsee">

    <label for="inputZipCode"><span>Postinumero</span></label>
    <input class="form-control"  type="text" name="zipCode" id="inputZipCode" placeholder="Osoitteen postinumero, esim '05200'">

    <label for="inputStartDate"><span>Aloitus:</span></label>
    <input class="form-control" type="date" name="startDate" id="inputStartDate" placeholder="Aloitus päivämäärä" required>

    <label for="inputEndDate"><span>Urakan valmistus päivä:</span></label>
    <input class="form-control" type="date" name="estimatedFinish" id="inputEndDate" placeholder="Urakan valmistumis päivämäärä" required>

    <label for="inputRate"><span>Urakan sopimus hinta</span></label>
    <input class="form-control" type="number" name="contractRate" id="inputRate" placeholder="Urakasta sovittu hinta" required>

    <input id="boolFalse" type="radio" name="progress" value="false" checked hidden>
    <input id="boolTrue" type="radio" name="progress" value="true" hidden>

    <br>
    <button class="submit" type="submit">Luo</button>
</form>
```

Kuva 14. HTML-koodilla luotu urakan luontilomake.

Syötteet varmistus pitää tehdä kuitenkin myös lähetyksen jälkeen, kun lomakkeen tiedot saapuvat palvelimelle. Varmistuksessa käytetään mongoose lisäosan tuomaa skeemaa (ks. luku 4.4.1). Mikäli vastaanotetut tiedot eivät vastaa skeeman vaatimia tietoja tai tietomuotoja, ei urakkaa hyväksytä ja käyttäjän urakkakohteen luontiyritys epäonnistuu. Kun vastaan otettavat tiedot täsmäävät skeeman vaatimia tietoja, ohjataan kohteen luontipyyntö urakkakohteista vastaavan ohjaimen `create`-metodiin. Tässä metodissa kaikki POST-pyynnön mukana tulleet parametrit tallennetaan yhteen muuttujaan avain-arvo tyylillä. Tietojen keräyksen jälkeen kutsutaan mongoose skeeman `Create()`-funktiota, mikä samanaikaisesti luo ja tallentaa uuden kohteen. Tietokannan ja sovelluksen välistä kommunikaatiota kuvataan paremmin luvussa 4.4.2. Kohteen luonnin jälkeen käyttäjä ohjataan urakoiden tarkastelu sivulle.

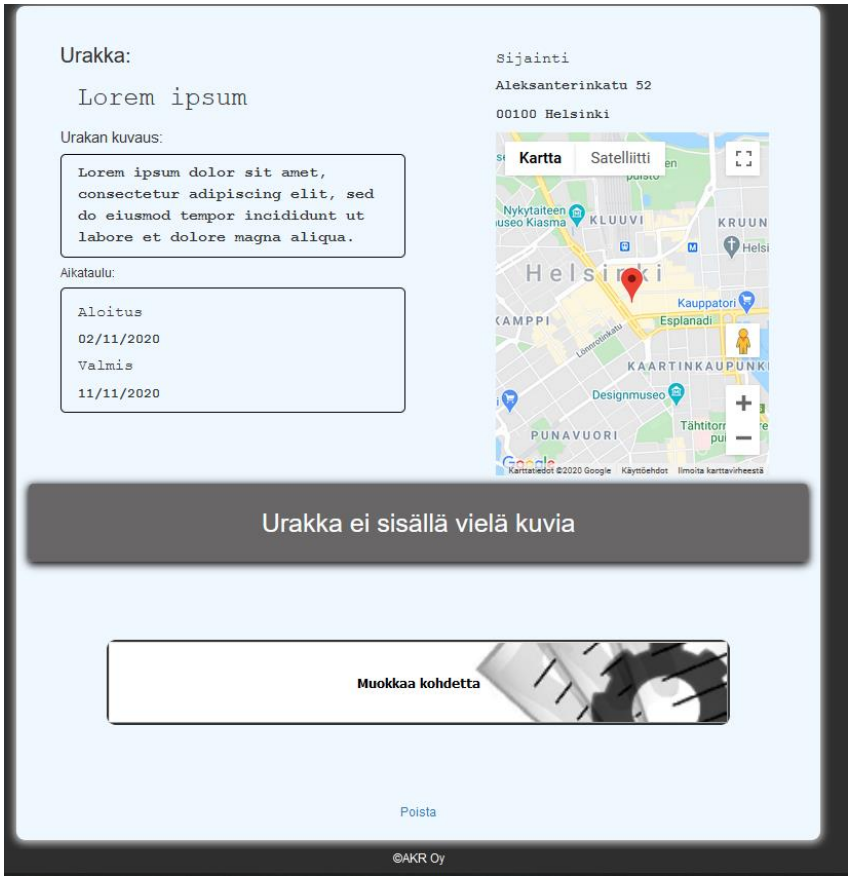
4.2.4 Kohteiden tarkastelu ja muokkaus

Käyttäjä on luonut uuden urakkakohteen ja haluaa nyt tarkastella kohdetta ja mahdollisesti muokata sitä. Tässä luvussa kuvataan sovelluksen toiminta urakoiden tarkastelussa, sekä niiden muokkauksessa.

Urakoiden tarkasteluun pääsee etusivun alimman painikkeen avulla tai navigointipalkissa olevasta ‘Urakat’ kohdasta. Näkymän ladattua käyttäjälle paljastuu lista tietokannasta löytyvistä urakoista. Mikäli sovellukseen ei ole vielä tallennettu urakoita, näkyy käyttäjälle teksti ‘Ei vielä tallennettuja urakoita.’. Jokaisesta listalla näkyvästä urakkakohteesta on kerrottu nimi, kuvaus, sijainti sekä urakan tila; valmis tai ei valmis. Listauksen alareunassa on nappula, jota painamalla urakkaa voi tarkastella tarkemmin. (Kuva 15.)

Kuva 15. Tallennettujen kohteiden lista näkymä.

Klikkaamalla alareunan nappia ‘Tarkastele urakkaa’ ohjataan käyttäjä näkymään, jossa on esillä urakkakohteen kaikki tiedot, sijainti kartalla sekä urakasta otetut kuvat. Tästä näkymästä tulee myös mahdolliseksi kuvien lisäys ja poisto. (Kuva 16.) (Liite 3.)



Kuva 16. Kohteen kaikkien tietojen esittäminen ja tarkastelu.

Esillä olevien muiden tietojen muokkaamiseksi, käyttäjän pitää klikata nappia ‘Muokkaa kohdetta’. Nappi ohjaa käyttäjän muokkausnäkymään, jossa kohteen tietoja muokataan täyttämällä vastaavanlainen lomake kuin kohteen luonnin yhteydessä.

Näkymässä oleva kartta toteutettiin Google Maps apin:n avulla ja kohteen sijainnin määritykseen käytettiin Googlen Geocodin api:n. Kartan toimintaan vaadittu koodi sijoitettiin map.js tiedostoon. Tiedostossa oleva funktio `initMap()` suoritetaan takaisinkutsufunktiona, kun yhteys Googlen Maps api:in on muodostettu onnistuneesti. Funktiossa alustetaan ensin sijainnin hakemiseen tarvittavat koordinaatit ja Geocoder, jonka avulla saadaan halutun osoitteen koordinaatit. Onnistuneen haun johdosta sivulle luodaan kartta, johon lisätään merkki osoittamaan kohteen sijainti. (Kuva 17.)

```
function initMap() {
    let lat;
    let lon;
    let geocoder = new google.maps.Geocoder();
    let address = document.getElementById('address').innerHTML;
    geocoder.geocode({
        'address': address
        }, function(results, status) {
            if(status == google.maps.GeocoderStatus.OK) {
                lat = results[0].geometry.location.lat();
                lon = results[0].geometry.location.lng();
                console.log("1 lat: " + lat + ", long: " + lon);
                let loc = {lat:lat, lng:lon};
                let map = new google.maps.Map(
                    document.getElementById('map'), {
                        zoom: 15,
                        center: loc
                    }
                );
                const marker = new google.maps.Marker({position: loc, map: map});
            }
        }
    );
}
```

Kuva 17. Karttanäkymän mahdollistava JavaScript koodi map.js tiedostossa.

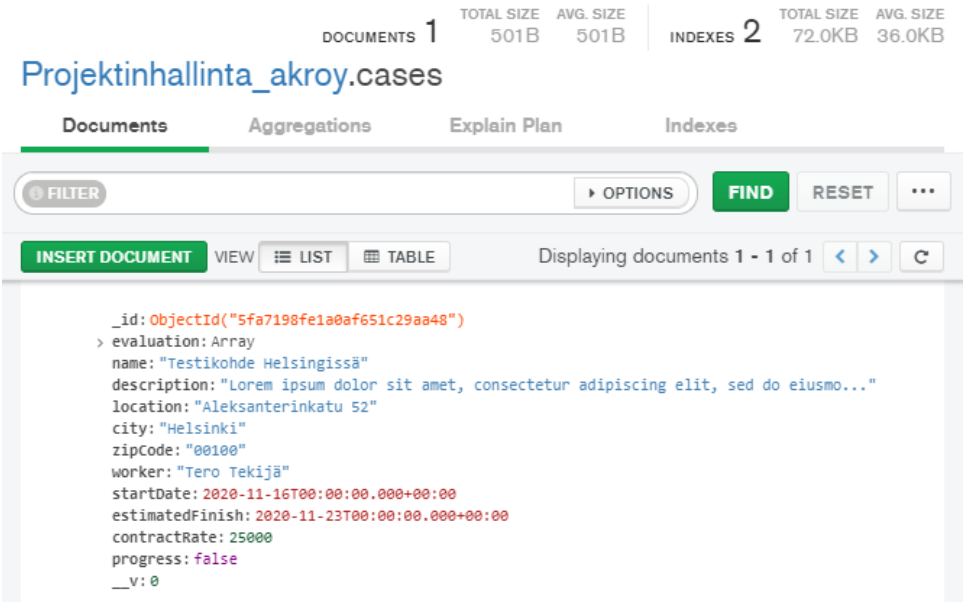
4.3 Tietokanta

Tässä työssä hyödynnettiin MongoDB tietokantaa. Kappaleessa käydään läpi MongoDB:n asennus kehitystyötä varten sekä yhteyden muodostaminen juuri asennettuun tietokantaan. Kappaleessa ei käydä läpi tietokannan pystyttämistä tuotantoympäristöön, mutta yhteys muodostetaan siten, ettei koodia tarvitse muuttaa tuotantoympäristöön siirryttäessä.

Asennus

Windows alustalle asennettaessa, MongoDB:n nettisivuilta pitää ladata asennusohjelma ja asentaa tietokanta asennusohjelman vakiovalintojen kautta. Asennuksen jälkeen tietokoneen C:-asemaan pitää luoda uusi kansio “data” ja tämän kansion sisään alikansio nimeltään “db”, jonne MongoDB tallentaa paikalliset tietokannat. Tietokannan saa käynnistettyä siirtymällä komentolinjalla MongoDB:n asennuskansioon ja ajamalla komennon ‘`mongod`’. Tietokantaan yhteyden muodostaminen onnistuu avaamalla uuden

komentolinjaikkunan samassa tiedostokansiossa ja ajamalla komennon ‘mongo’ Vaihtoehtoisesti tietokannan voi käynnistää ja sitä voi hallinnoida asennuksen yhteydessä asentuneen MongoDB Compass sovelluksen avulla. Compass sovellus tarjoaa helppokäyttöisen graafisen käyttöliittymän tietokannan ylläpitoon. (Kuva 18.)



Kuva 18. MongoDB Compass sovelluksen näkymä projektin urakoista.

Tietokannan lisäys ja yhteyden muodostus node.js-ympäristöön

Yhteyden muodostaminen Node.js-ympäristössä hoituu näppärästi Mongoose-lisäosan avulla. Kehitystyössä joudutaan määrittelemään tietokannan osoite, mutta tuotantoympäristössä tietokannan osoite saadaan Node.js:n sisäisistä prosesseista, siksi yhdistämisessä on käytetty ‘| |’ -operaattoria. Mikäli tuotantoympäristön osoitetta ei löydy, käytetään vakio-osoitetta eli kehitysympäristön osoitetta. Yhteyttä muodostaessa varmistetaan lippujen avulla, ettei yhteyden muodostaessa käytetä vanhentuneita tapoja tietokannan kanssa kommunikointiin. (Kuva 19.)

```
const dbURL = "mongodb://localhost:27017";
const dbName = "Projektinhallinta_akroy";
mongoose.connect(`${(process.env.MONGODB_URI || dbURL + "/" + dbName)}`, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    useCreateIndex: true,
});
```

Kuva 19. Yhteyden luonti tietokantaan.

4.3.1 Tietomallit

Mongoose auttaa tietokannan ylläpidossa helpottamalla tietojen syöttöä ja rakentamista. Mongoose:n avulla voidaan luoda tietokantaan malleja, joiden avulla määritellään mitä ja minkälaisessa muodossa tietokannan objektien sisältämä tieto pitää olla. Nämä mallit eli skeemat helpottavat huomattavasti tietokannan ja sovelluksen välistä toimintaa. Työssä käytetään kolmea eri skeemaa tiedon tallentamiseen; käyttäjä, urakkakohde ja kuntoarvio. Käyttäjä skeemaan lisätään ‘passport-local-mongoose'-lisäosa, jolla mahdollistetaan käyttäjien autentikointi Passport.js väliohjelman avulla.

Mallien määrittelyssä pitää miettiä mitä tietoa mallit tarvitsevat. Käyttäjämallin kohdalla vaaditaan käyttäjänimi ja salasana. Käyttäjätunnus tulee olla uniikki, ettei järjestelmään voida lisätä useita saman nimisiä käyttäjiä. Malliin sisällytetään myös mahdollisuus nimetä käyttäjä, mutta tästä tiedosta ei tehdä pakollista vielä kenttää, pääosin asiakkaan toiveiden toimesta. Jotta voidaan mahdollistaa järjestelmänvalvojantilin luonti, lisätään skeemaan totuusarvo, jossa arvo tosi merkitsee valvojan oikeuksia. Lisäksi käyttäjille annetaan `‘timestamp’` arvo, mikä kertoo käyttäjäprofiilin luontiajankohdan. Kuvassa 20 havainnollistetaan käyttäjän tiedoille luotua tietomallia.

```
{ Schema } = mongoose,
userSchema = new Schema(
    {
        name: {
            type: String,
            trim: true
        },
        username: {
            type: String,
            required: true,
            lowercase: true,
            unique: true
        },
        a: Boolean

    },
    {
        timestamps: true
    }
);
```

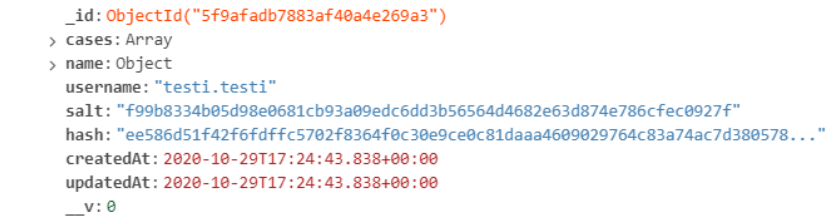
Kuva 20. Käyttäjätietojen malli, joka määrittelee tietokantaan lisättävän käyttäjän tiedot JSON muodossa.

Käyttäjien autentikointia varten malliin lisätään Mongoose:n `plugin`-metodilla `passportLocalMongoose`-lisäosa, jotta Passport.js osaa käyttää käyttäjän tunnusta kirjautumisessa. (Kuva 21.)

```
userSchema.plugin(passportLocalMongoose, {
    usernameField: "username"
});
```

Kuva 21. Käyttäjän autentikointi Passport.js:n avulla

Salasanaa ei tarvitse erikseen määritellä käyttäjän skeemaan, sillä Passport.js tunnistaa automaattisesti salasana kentän ja hoitaa salauksen sovelluksen taka-alalla automaattisesti. Tämä tarkoittaa sitä, ettei sovellus missään vaiheessa tiedä käyttäjän salasanaa selkotekstinä. Kuvassa 22 esitetään, miten tiedot näkyvät tietokannassa.



Kuva 22. Käyttäjän tiedot esitetty tietokannassa.

Urakka ja kuntoarvioiden skeemat luodaan käyttäjäskeeman tavoin, erona käyttäjään on, ettei urakka- tai kuntoarviomallien arvoihin tarvitse lisätä salausta. Mongoose sallii mallien välisiä yhteyksiä, joita urakkamallissa hyödynnetään. Urakoihin lisätään kuntoarvioille kenttä, johon asetetaan viittaus kuntoarviomalliin, jotta kuntoarviot näkyvät vain sen tehdyn urakan yhteydessä. Yhteys mallien välillä tarvitsee tehdä vain yhteen malliin. (Kuva 23.)

```
{ Schema } = mongoose,
caseSchema = new Schema({
    name: {
        type: String,
        required: true,
        unique: true,
        trim: true
    },
    description: {
        type: String,
        required: true,
        trim: true
    },
    location: {
        type: String,
        trim: true
    },
    city:{
        type: String,
        trim: true
    },
    zipCode: {
        type: Number,
        min: [10000, "Postinumero liian lyhyt"],
        max: 99999,
    },
    startDate: {
        type: Date,
        required: true
    },
    estimatedFinish: {
        type: Date,
    },
    contractRate: {
        type: Number,
        min: [0, "Hinta ei voi olla negatiiviinen"],
        max: [9999999]

    },
    progress: Boolean,
    evaluation: [{type: Schema.Types.ObjectId, ref: "Evaluation"}],
    images: [{
        data: Buffer,
        contentType: String,
        desc: {type: String}
        }
    ]
});
```

Kuva 23. Urakkakohteen skeema.

4.3.2 CRUD

Sovelluksessa noudatetaan CRUD-toimintamallia jokaisen tietokannassa olevan kohteen kanssa. CRUD-lyhenne tulee sanoista create, read, update ja delete [5, s. 220]. Nämä toiminnot sijaitsevat ‘controllers’-kansiossa, jossa toiminnot on jaettu eri ohjaimien välille. Jokainen ohjain sisältää yhden tietomallin toiminnot tietokannan väliseen kommunikointiin. Näihin toimintoihin kuuluu mallin luonti (create), tietojen luku (read), tietojen päivitys (update) ja poisto (delete).

Create – luonti

Internetin maailmassa tietokantoihin luodaan uutta tietoa HTML:n POST-metodi käyttäen. Useimmiten luonti tapahtuu täytettävien lomakkeiden avulla, niin se tapahtuu myös tässä työssä. Sovelluksessa luodaan uutta tietoa tietokantaan HTML:n lomakkeilla ja nämä lomakkeet syötetään tietokantaan POST-metodilla, jotta sovellus tietää tallentaa tiedot.

Kun käyttäjä luo sovellukseen uuden urakkakohteen, täyttää hän sovelluksessa olevan lomakkeen. Kun lomake lähetetään, se lähetetään POST-metodilla uuden urakan tallennuspolkuun. Lomakkeen saapuessa palvelin ohjaa pyynnön routes.js-tiedostoon, jossa palvelin suorittaa halutun toiminnon pyynnön metodin ja polun perusteella. Urakkakohteen luonnissa, palvelin suorittaa urakoiden ohjaimessa olevan create-operaation, jossa tiedot kerätään pyynnöstä, verrataan urakkamalliin ja tallennetaan tietokantaan, mikäli urakan sisältämät tiedot noudattavat urakkamallia. Kuvassa 24 näytetään koodi, joka vastaa uuden urakkakohteen lisäämisestä tietokantaan.

```
create: (req, res, next) => {
    let caseDetails = {
        name: req.body.name,
        description: req.body.description,
        location: req.body.address,
        city: req.body.city,
        zipCode: req.body.zipCode,
        startDate: req.body.startDate,
        estimatedFinish: req.body.estimatedFinish,
        contractRate: req.body.contractRate,
        progress: req.body.progress
    };
    caseModel.create(caseDetails)
    .then(casemodel => {
        res.locals.redirect = "/all";
        res.locals.user = casemodel;
        next();
    })
    .catch(error => {
        console.log(`error saving case ${error.message}`);
        next();
    });
},
```

Kuva 24. Uuden urakan luonti tietokantaan.

Read – luku

Tietojen lukemisessa palvelimelle lähetetään GET-metodilla pyyntö, tiedon tarkastelun merkiksi. Sovelluksessa luetaan tietokannan tietoja urakoita tarkastellessa. Kun käyttäjä haluaa tarkastella kaikkia urakoita, siirtyy hän urakat-sivulle jolloin palvelin saa pyynnön hakea kaikki tietokantaan tallennetut urakat. Palvelin kutsuu urakoiden ohjaintiedostosta oikeaa operaatiota ja tietokannasta haetaan kaikki urakat `find()`-funktion avulla. Löydetyt urakat lähetetään takaisin sovellukselle käyttäjän nähtäväksi. (Kuva 25.)

```
getCases: (req, res, next) => {

    caseModel.find({})
    .exec()
    .then(cases => {
        console.log(cases);
        res.render("cases/all", {
            cases: cases
        });
    })
    .catch(error => {
        res.render("error");
    })
```

Kuva 25. Kaikkien urakoiden haku tietokannasta.

Update - päivitys

Tiedon päivittämiseksi tietokannassa käytetään HTML:n PUT-metodia. Noden kanssa ongelmaksi muodostuu se ettei node.js ymmärrä luonnostaan PUT-metodia html:n lomakkeiden lähetyksessä. Ratkaisuja tämän ongelman selvittämiseksi on monia, mutta tässä työssä ongelma ratkaistaan sovellukseen asennettavan ’method-override’-paketin avulla. Paketilla HTML-pyyntöön voidaan lisätä parametri, joka kertoo palvelimelle mitä metodia lomakkeen lähetyksen yhteydessä tulee käyttää pyynnön alkuperäisen metodin sijasta. Kuvat 26 ja 27 näyttävät palvelinpuolen koodin sekä sovelluksen HTML-koodin parametrien havaitsemiseen.

```
router.use(methodOverride("_method", {
    methods: ["POST", "GET"]
}));
```

Kuva 26. Metodin märittely HTML:n parametrina

```
<form method="POST" action="<%=`/cases/${casemodel._id}/update?_method=PUT`%>">
    <h2>Urakan muokkaus:</h2>
```

Kuva 27. POST-metodin määrittely muokkauspyynnön parametreissa PUT-metodiksi.

Päivitys toimii jälleen työn jokaisen tietokannan kohteella samalla periaatteella. Jotta muokatun tiedon saa lisättyä tietokantaan, pitää kohteen uudet tiedot ensin kerätä yhteen, jonka jälkeen tietokannasta pitää löytää oikea kohde. Tietojen päivitys onnistuu kuitenkin Mongoose:n avulla yksinkertaisesti. Päivitettävät tiedot kulkeutuvat lomakkeen yhteydessä palvelimelle, jossa tiedot rakennetaan yhden muuttujan sisälle ja '`findByIdAndUpdate()`'-funktio suorittaa tietojen päivityksen tietokantaan. (Kuva 28.)

```
update: (req, res, next) => {
    let caseId = req.params.id;
    let caseParams = {
        name: req.body.name,
        description: req.body.description,
        location: req.body.address,
        zipCode: req.body.zipCode,
        startDate: req.body.startDate,
        estimatedFinish: req.body.estimatedFinish,
        contractRate: req.body.contractRate,
        progress: req.body.progress

    };
    caseModel.findByIdAndUpdate(caseId, {
        $set: caseParams
    })
        .then(casemodel => {
            res.locals.redirect = `/case/${caseId}`;
            res.locals.casemodel = casemodel;
            next();
        })
        .catch(error => {
            console.log(`Error while updating case: ${error.message}`);
            next(error);
        });
},
```

Kuva 28. Urakkakohteen päivitys tietokantaan.

Delete - poisto

Lisäämisen, muokkaamisen ja lukemisen lisäksi yksi tärkeä ominaisuus tietokannan käsittelyssä on mahdollisuus myös poistaa sinne lisättyä tietoa. Poisto mahdollisuus täytyy kuitenkin toteuttaa siten, ettei tietoja poisteta kannasta vahingossa. Tätä varten ennen poistoa on kysyttävä käyttäjältä varmistus kohteen poistamisesta ennen itse toiminnon suorittamista. Yksinkertainen varmistus kysely ennen komennon toteuttamista voidaan

toteuttaa HTML koodin ‘onclick’-tapahtumassa. Toisin sanoen, kun ruudulla olevaa ‘poista’ nappia klikataan, kysytään käyttäjältä heti varmistus toiminnon loppuun suorittamisesta. (Kuva 29.)

```
<a href="<%= `${c._id}/delete?_method=DELETE` %>"
    onclick="return confirm('Haluatko varmasti poistaa kohteen?')">
    Poista
</a>
```

Kuva 29. Esimerkki html koodista, urakkakohteen poiston yhteydessä.

Tietojen poistoa tietokannasta kutsutaan käyttämällä DELETE-metodia. Tietojen poisto onnistuu Mongoosen avulla nopeasti. Etsimällä parametrien yhteydessä tulleella kohteen tunnisteella oikea kohde tietokannasta, voidaan löydetty kohde poistaa ‘findByIDAndRemove’-funktion avulla. (Kuva 30.)

```
delete: (req, res, next) => {
    let caseId = req.params.id;
    caseModel.findByIdAndRemove(caseId)
        .then(() => {
            res.locals.redirect = "/all";
            next();
        })
        .catch(error => {
            console.log(`Error while deleting user by ID: ${error.message}`);
            next();
        });
},
```

Kuva 30. Palvelimella suoritettava koodi urakkakohdetta poistaessa.

5 YHTEENVETO

Opinnäytetyön tavoitteena oli luoda helppokäyttöinen sovellus yrityksen urakoiden seurantaan. Opinnäytteessä keskityttiin sovelluksen vaatimusten keräämiseen ja sovelluksen luontiprosessiin.

Raportin kirjoittamisen aikana sovellus noudattaa tärkeimpiä asiakkaan vaatimuksia, jotka olivat uusien urakkakohteiden luonti ja niiden tarkastelu, sekä helppokäyttöisyys. Sovellus sisältää helppokäyttöisen käyttöliittymän sekä mahdollistaa uusien urakkakohteiden luomisen ja niiden tarkastelun. Vaikka käyttöliittymä ei ole jokaisessa sovelluksen osa-alueessa täysin valmis, on asiakas ilmaissut tyytyväisyytensä sovelluksen valmiina oleviin osiin.

Työn aikana kohtasin paljon haasteita ohjelmistoprojektin kehitykseen liittyen. Omien ja toimeksiantajan kiireiden vuoksi työn aloitus myöhästyi useita viikkoja. Kun ensimmäinen asiakastapaaminen saatiin vihdoin sovittua, vastaan tuli kielimuuri. Asiakas ei osannut kertoa tarkasti sovelluksen haluttua toimintaa, sillä asiakkaalla ei ollut tietoa siitä, mitä oikeasti kaipasi. Pitkän asiakastapaamisen päätteeksi onnistuin kuitenkin selvittämään minkälaista sovellusta oli tarkoitus lähteä kehittämään. Idean luonnin jälkeen vaatimustenkin keräys onnistui hyvin.

Teknologioiden valintaan vaikutti paljon aikaisempi kokemus kyseisten teknologioiden kanssa, sillä täysin tuntemattomien teknologioiden opettelu olisi hidastanut työn edistymistä entisestään. Kokemusta työhön valittuihin teknologioihin ei ollut kerääntynyt paljoa, mutta perusteet oli tullut tutuksi lähes jokaisesta koulussa käytyjen kurssien kautta.

Sovelluksen kehitystyö alkoi luontevasti suunnittelemalla ensin sovelluksen rakennetta, kuten mitä tietoa sovellukseen pitää tallentaa ja missä muodossa sekä miten navigointi sovelluksen eri osissa onnistuisi. Ohjelmointi osuus sisälsi paljon onnistumisia kuin myös pitkiä valvottuja öitä ohjelmointivirheiden korjaamisessa.

Kehitystyön aikana esittelin sovellusta muutamien viikkojen välein asiakkaalle tai vasta kun sovellukseen oli tullut esittämisen arvoista edistystä. Kehitystyön aikana olisin kaivannut asiakkaalta enemmän palautetta ja ajatuksia kehitettävien ominaisuuksien osalta, sillä sain tekoon täysin vapaat kädet ja ulkopuoliset mielipiteet olisivat auttaneet kehitystyössä. Vaatimusten perusteella luodut käyttötapaukset ja käyttötapauskaaviot nopeuttivat ja helpottivat käyttöliittymän luontia.

Tämänhetkinen sovellus sopii yrityksen tarpeisiin, mutta vaati vielä jatkokehitystä. Toimeksiantaja testaa sovelluksen toimintaa ja mikäli sovelluksen toiminta vaikuttaa lupaavalta, viedään kehitystä eteenpäin, jolloin myös hiotaan tietoturvaa entistä turvallisemmaksi. Asiakas on myös ilmaissut halukkuutensa viedä sovellus kaupalliselle tasolle, mikäli testausvaihe on lupaava ja jatkokehitys onnistuu. Tulevaisuuden kannalta jää nähtäväksi, kuinka pitkälle sovelluksen kehitys aiotaan todellisuudessa viedä.

LÄHTEET

1. Harri Laine, History of SQL, Helsingin Yliopisto, n. d. https://www.cs.helsinki.fi/u/laine/tuelip/sql_material/sql_history.html. Luettu 5.9.2020.
2. GitHub features, 2020. GitHub. https://github.com/features Luettu 6.9.2020.
3. GitHub Pricing, 2020. GitHub. https://github.com/pricing Luettu 6.9.2020.
4. FAQ about changes to GitHub's plans, 2020. GitHub. https://docs.github.com/en/github/getting-started-with-github/faq-about-changes-to-githubs-plans#what-plans-and-pricing-changes-did-github-announce-on-april-14 Luettu 6.9.2020.
5. Jonathan Wexler, Get Programming with Node.js, 2019, Manning Publications.
6. MongoDB, 2020. MongoDB https://www.mongodb.com Luettu 5.9.2020.
7. The Node.js Event-loop, Timers and process.nextTick(), 2020. Node.js. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/. Luettu 5.9.2020.
8. James Heumann, Tips for writing good use cases, 2008, IBM. http://uclab.khu.ac.kr/lectures/2015_1_sw/Practice_Text_01.pdf Luettu 29.10.2020.
9. What is npm?, 2011. Node.js. https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/ Luettu 29.10.2020.
10. What is Use Case Diagram, Visual Paradigm, n.d. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ Luettu 2.11.2020.
11. Evan M.Hahn, Express in Action – Writing, building, and testing Node.js applications, 2016, Manning Publications.

Käyttötapaus – Uuden kuntoarvion luonti

TUNNISTE

id_1

KUVAUS

Käyttäjä haluaa luoda järjestelmään uuden kuntoarvion.

Toimija(t)

Sovelluksen käyttäjä

Ennakkoehdot

1. Sovelluksen käyttäjä on kirjautunut onnistuneesti järjestelmään.
2. Sovelluksen käyttäjä on sovelluksen etusivulla.

Tapahtumien kulku

1. Käyttäjä klikkaa ruudulla näkyvää nappia 'Luo uusi kuntoarvio'.
2. Käyttäjälle avautuu näkymä, jossa näkyy kuntoarvion luomiskaavio.
3. Käyttäjä täyttää lomakkeen vaatimat tiedot.
4. Käyttäjä klikkaa lomakkeen alareunassa olevaa nappia 'Luo'.
5. Uusi kuntoarvio tallentuu tietokantaan.

Jälkiehdot

Käyttötapaus ei sisällä jälkiehtoja

Käyttötapaus – Uuden urakkakohteen luonti

TUNNISTE

id_2

KUVAUS

Käyttäjä haluaa luoda järjestelmään uuden urakkakohteen.

Toimija(t)

Sovelluksen käyttäjä

Ennakkoehdot

1. Sovelluksen käyttäjä on kirjautunut onnistuneesti järjestelmään.
2. Sovelluksen käyttäjä on sovelluksen etusivulla.

Tapahtumien kulku

1. Käyttäjä klikkaa ruudulla näkyvää nappia 'Luo uusi urakkakohde'.
2. Käyttäjälle avautuu näkymä, jossa näkyy urakkakohteen luomislomake.
3. Käyttäjä täyttää lomakkeen vaatimat tiedot.
4. Käyttäjä klikkaa lomakkeen alareunassa olevaa nappia 'Luo'.
5. Uusi urakkakohde tallentuu tietokantaan.

Jälkiehdot

Käyttötapaus ei sisällä jälkiehtoja

Käyttötapaus – Tallennettujen urakkakohteiden tarkastelu

TUNNISTE

id_3

KUVAUS

Käyttäjä haluaa tarkastella sovellukseen tallennettujen urakoiden tietoja.

Toimija(t)

Sovelluksen käyttäjä

Ennakkoehdot

1. Sovelluksen käyttäjä on kirjautunut onnistuneesti järjestelmään.
2. Sovelluksen käyttäjä on sovelluksen etusivulla.
3. Järjestelmään on tallennettu vähintään yksi urakkakohde.

Tapahtumien kulku

1. Käyttäjä klikkaa sovelluksen ylimmän navigointipalkin nappia 'Urakat' tai etusivulla näkyvää nappia 'tarkastele urakoita'.
2. Käyttäjälle aukeaa näkymä, jossa on listattu kaikki tallennetut urakkakohteet.
3. Kohteet on listattu allekkain näkymässä ja jokaisella kohteella on nappula, josta kohdetta voi tarkastella.
4. Käyttäjä klikkaa 'tarkastele urakkaa' nappia.
5. Käyttäjälle aukeaa näkymä, jossa urakkakohteen tiedot ovat tarkasteltavissa.

Jälkiehdot

Käyttötapaus ei sisällä jälkiehtoja

Käyttötapaus – Kuvien lisääminen urakkakohteesta

TUNNISTE

id_4

KUVAUS

Käyttäjä haluaa tallentaa kuvia urakkakohteesta ja kirjoittaa kuvasta lyhyen kuvauksen, mikä näkyy kuvan alapuolella.

Toimija(t)

Sovelluksen käyttäjä

Ennakkoehdot

1. Sovelluksen käyttäjä on kirjautunut onnistuneesti järjestelmään.
2. Sovelluksen käyttäjä on sovelluksen etusivulla.
3. Järjestelmään on tallennettu vähintään yksi urakkakohde.

Tapahtumien kulku

1. Käyttäjä klikkaa sovelluksen ylimmän navigointipalkin nappia 'Urakat' tai etusivulla näkyvää nappia 'tarkastele urakoita'.
2. Käyttäjälle aukeaa näkymä, jossa on listattu kaikki tallennetut urakkakohteet.
3. Kohteet on listattu allekkain näkymässä ja jokaisella kohteella on nappula, josta kohdetta voi tarkastella.
4. Käyttäjä klikkaa 'tarkastele urakkaa' nappia.
5. Käyttäjälle aukeaa näkymä, jossa urakkakohteen tiedot ovat tarkasteltavissa.
6. Kohteen tietojen alapuolella on nappula jota klikkaamalla aukeaa valintaikkuna kuvien lisäämiselle.
7. Käyttäjä valitsee päätelaitteesta lisättävän kuvan.
8. Kuva näkyy käyttäjälle valintaikkunassa.
9. Kuvan alla on tekstilaatikko, johon käyttäjä kirjoittaa halutun tekstin kuvasta.
10. Käyttäjä klikkaa nappia 'tallenna'.
11. Kuva tallentuu urakkakohteen tietoihin.

Jälkiehdot

Käyttötapaus ei sisällä jälkiehtoja