JAVA-SOVELLUKSEN MUUTTAMINEN PILVIPALVELUSOVELLUKSEKSI

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Hämeen ammattikorkeakoulu

Tietojenkäsittelyn koulutusohjelma

HAMK Visamäki syksy 2016

Aleksi Rinta-Kiikka
TRTKNU13A3

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

VISAMÄKI
Tietojenkäsittely
Systeemityö

Tekijä Aleksi Rinta-Kiikka **Vuosi** 2016

Työn nimi Java-sovelluksen muuttaminen pilvipalvelusovellukseksi

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli toteuttaa toimeksiantajan verkkosovellus uudelleen käyttäen uusia nykypäivän JavaScript-tekniikoita, kuvata sen muutos pilvipalvelusovellukseksi ja selvittää tämän toteutustavan hyödyt. Käytetyiksi tekniikoiksi ja teknologioiksi valikoitui AngularJS, Node.js ja OpenShift. Työn toimeksiantajana toimi Ambientia Group Oy, joka on Hämeenlinnasta lähtöisin oleva IT-alan yritys. Työn tavoitteisiin päästiin ja tuotoksena syntyi lähes julkaisuvalmis pilvipalvelusovellus, joka miellytti itse tekijää sekä toimeksiantajaa. Työ aloitettiin tutustumalla valittuihin tekniikoihin verkosta löytyneiden materiaalien ja tutoriaalien avulla, jonka jälkeen aloitettiin toteuttamaan itse sovellusta.

Tekijälle oli koulutöiden ja -projektien ansiosta kertynyt kokemusta verkkosovellusten kehityksestä, joten ohjelmointitaidot olivat suhteellisen vakaalla pohjalla ennen työn aloitusta. Opinnäytetyössä käytetyt teknologiat olivat kuitenkin täysin uusia, joka teki työn toteuttamisesta haastavaa, mutta myös mielenkiintoista.

Työ on jaettu teoria- ja käytännönosuuteen. Teoriaosuudessa käsitellään toimeksiantajan alkuperäisen sovelluksen taustaa ja toimintaa sekä AngularJS:n, Node.js:n ja OpenShiftin taustaa, pääkonsepteja ja hyötyjä. Käytännönosuudessa käydään läpi uuden sovelluksen päätoimintojen toteutusta sekä niiden toimintaa osana sovellusta.

Opinnäytetyötä tehdessä todettiin, että AngularJS ja Node.js ovat erittäin hyviä valintoja dynaamisen verkkosovelluksen kehittämiseen ja ovat pienellä vaivalla helposti omaksuttavissa. OpenShift todettiin pilviympäristönä erittäin hyvänä ja helppokäyttöisenä. Ilmenneistä ongelmista selvittiin kattavien dokumentointien ja koodiesimerkkien avulla, joita pystyttiin soveltamaan työssä. Työn tavoitteisiin päästiin, vaikka sovellusta ei saatu työn aikana julkaisuvalmiiksi asti. Sovellukseen saatiin kuitenkin implementoitua kaikki suunnitellut toiminnot ja ominaisuudet.

Avainsanat Node.js, AngularJS, OpenShift, verkkosovelluskehitys

Sivut 36 s.

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

VISAMÄKI
Business Information Technology
System development

Author Aleksi Rinta-Kiikka **Year** 2016

Subject of Bachelor's thesis
Turning a Java application into a cloud based application

ABSTRACT

The goal of this Bachelor's thesis was to recreate the client's web application using new JavaScript technologies, illustrate its transformation into a cloud based application and report the benefits. The technologies that were chosen to be used in this work were AngularJS, Node.js and OpenShift. The commissioner and client of this thesis was an IT company from Hämeenlinna called Ambientia Group Oy. The goals of this thesis were achieved and as a result a new version of the web application was made that was almost ready to be published. The result satisfied the author of this thesis and the client. The project started by getting familiar with the chosen technologies by going through documentation and code examples that were found in the web.

The author of this thesis had gained some experience in web application development from school projects so his programming skills were on a solid base before starting the project. The technologies used in this project were completely new which made it challenging, but also interesting.

The thesis is divided into two parts: theory and practical part. The theory part goes through the background and main features of the client's application and also the background, main concepts and benefits of AngularJS, Node.js and OpenShift. The practical part goes through the development of the main features of the new application and how they work as a part of the application.

During the process of this thesis, AngularJS and Node.js were found to be good choices when developing dynamic web applications. They were also found easy to adopt to after a little work. OpenShift was found to be very easy to use and a good cloud environment to run a web application in. The difficulties faced in this project were solved with very vast documentations and code examples that were possible to be applied into the project. The goals were achieved even though the new version wasn't published by the end of this thesis. All the planned main features were implemented.

Keywords Node.js, AngularJS, OpenShift, web application development

Pages 36 p.

Termistö

Sovelluskehys on kirjasto, jonka avulla on mahdollista tukea tai ohjata sovelluksen rakentamista (engl. framework).

DOM on tapa kuvata rakenteisen dokumentin rakenne puuna, jonka osia voidaan hakea tutkia ja manipuloida (engl. Document Object Model).

Template on dokumentti, jonka pohjalta selain koostaa verkkosivun.

Asynkroninen eli ei-reaaliaikainen. Jokin joka toimii prosessin taustalla tukkimatta sitä.

Monoliittinen sovellus on suuri useista erillisistä osista koostuva saumattomasti ohjelmoitu sovellus.

Skeema määrittää tietokannan kenttien nimet, tietotyypit, rajoitteet ja ominaisuudet.

Pilvipalvelusovellus on sovellus, joka toimii pilviympäristössä eli jonkin palveluntarjoajan palvelimilla.

Funktio on lohko koodia, joka suorittaa tietyn toiminnon tai toimintoja.

Parametri on arvo, joka voidaan antaa funktiolle.

Objekti on olio, joka voi sisältää muuttujan, funktion tai listan tai taulukon, joka sisältää useita arvoja. Objekti voi sisältää myös näitä kaikkia.

Muuttuja on symboli, joka edustaa jotakin tiettyä arvoa.

Kysely on tietokantaan lähetettävä komento, jolla voidaan hakea tietoja tietokannasta.

Interpolointi tarkoittaa, jonkin arvon lisäystä jo määritettyjen merkkien väliin.

Tagi on HTML-kielessä käytettävä elementtien merkkaustapa.

HTTP on protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon (engl. Hypertext Transfer Protocol).

JavaScript on ohjelmointikieli, jolla voidaan tehdä vuorovaikutteisia toimintoja sivulle.

HTML on avoimesti standardoitu kuvauskieli, nettisivujen rakentamiseen (engl. Hypertext Markup Language).

Hash-koodi on satunnaisista merkeistä ja numeroista koostuva merkkijono.

URL on verkko-osoite, jolla viitataan verkosta löytyvään resurssiin (engl. Uniform Resource Locator).

SISÄLLYS

1 JOHDANTO

Tämän aiheen tarjosi Ambientia Group Oy, joka on Hämeenlinnasta lähtöisin oleva IT-alan yritys. Ambientia on vuonna 1996 perustettu konsultointi, palvelumuotoilu ja sovelluskehitys yritys, joka tuottaa verkko- ja mobiilisovelluksia asiakkaiden tarpeisiin. (Ambientia 2016).

Työn taustalla on toimeksiantajan ja tekijän mielenkiinto uusien sovelluskehitystekniikoiden mahdollisuuksiin sekä niihin tutustumiseen ja niiden opiskeluun. Työn tekijällä on kokemusta verkkosovellusten ohjelmoinnista ainoastaan opintojen osalta muutaman projektin verran. Työssä käytettävien teknologioiden osalta tekijällä ei ole yhtään kokemusta. Tekijän motivaationa toimii opintojen myötä herännyt mielenkiinto ohjelmointiin sekä halu oppia uutta ja kehittää itseään.

Työn tarkoituksena on selvittää ja kuvata miten yhdellä palvelimella toimiva, Javalla ohjelmoitu monoliittinen sovellus voidaan toteuttaa käyttämällä erilaisia nykypäivän JavaScript-kirjastoja ja -kehyksiä. Työssä kuvataan myös toimeksiantajan sovelluksen uudelleentoteutus JavaScript-pilvipalvelusovellukseksi ja toteutustavan hyödyt. Työn tuotoksia tulee olemaan edellä mainitun kaltainen uusi versio toimeksiantajan sovelluksesta ja raportti sovelluksen toteutuksessa käytettyjen teknologioiden pääkonsepteista, hyödyistä ja toiminnasta sekä kuvaus uuden sovelluksen päätoimintojen toteutuksesta opinnäytetyössä.

Sovelluksen uusi versio toteutetaan AngularJS ja Node.js teknologioiden avulla Red Hatin OpenShift Online -pilvialustalle. Opinnäytetyön teoriaosuuden alussa käydään läpi toimeksiantajan alkuperäisen sovelluksen toiminta ja käyttötapaukset. Tämän jälkeen tutustutaan uuden version toteutuksessa käytettyjen teknologioiden pääkonsepteihin sekä niiden toimintaan verkkosovellusten kehityksessä. Käytännönosuudessa kuvaillaan sovelluksen päätoimintojen toteutus kooditasolla. Osuudessa ei tulla käymään läpi sovelluksen toiminnan kannalta vähemmän merkittäviä aspekteja, kuten ulkoasua. Käytännönosuuden pohjalta ei ole tarkoitus pystyä toteuttamaan samankaltaista sovellusta, vaan sitä on tarkoitus lukea projektin rinnalla havainnollistavana tukimateriaalina ja antaa käsitys miten valittuja teknologioita voidaan hyödyntää keskenään verkkosovellusta kehittäessä. Opinnäytetyön lopussa käydään läpi tehdyn työn tulokset ja kuvaillaan uuden sovelluksen hyödyt verrattuna alkuperäiseen versioon.

Opinnäytetyön tutkimuskysymyksiä ovat:

- Mitä ovat AngularJS, Node.js ja OpenShift?
- Miten toteuttaa pilvipalvelusovellus näiden teknologioiden avulla?
- Mitkä ovat tämän toteutustavan hyödyt?

2 ALKUPERÄINEN SOVELLUS

Alkuperäinen sovellus on niin sanottu latauslinkkigeneraattori. Sovellus toimii siten, että käyttäjä voi ladata tiedoston palvelimelle, jonka jälkeen sovellus generoi linkin, jota kautta tiedoston pääsee lataamaan. Sovellus on toteutettu monoliittisesti Java-ohjelmointikielellä. Tässä luvussa kerrotaan Javasta ja Java-ohjelmointikielestä sekä sen höydyistä ja haitoista kehittäessä sovellusta. Tämän jälkeen käydään läpi alkuperäisen sovelluksen toiminta ja käyttötapaukset.

2.1 Java

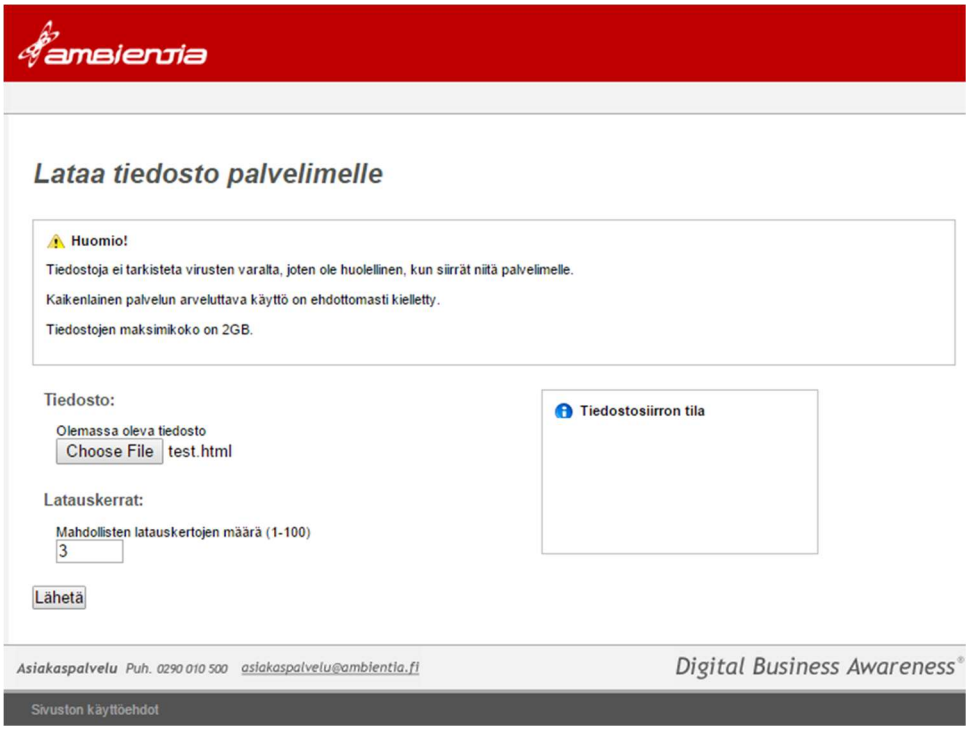
Java on ohjelmointikieli ja alusta, jonka julkaisi Sun Microsystems vuonna 1995. Nykyään on olemassa jo valtava määrä sovelluksia ja internetsivustoja, jotka eivät toimi, ellei Javaa ole asennettuna. (Oracle n.d.a.)

Java on suunniteltu tehokkaaksi, mutta myös mahdollisimman yksinkertaiseksi verrattaessa muihin samankaltaisiin olio-painotteisiin ohjelmointikieliin. Olio-ohjelmointia ymmärtävän kehittäjän on erittäin helppo ottaa Java käyttöön. Javalla ohjelmoidessa säästyy myös huomattavasti vähemmällä koodin määrällä kuin esimerkiksi C++-kielellä ohjelmoitaessa. (Oracle n.d.b.) Java pyrkii välttämään virhealttiita tapahtumia ja toimintaa painottamalla koodin ajamisen ja sen aikana tapahtuvien virheiden tarkistamista, joka lisää vakautta ja turvallisuutta. Java on monisäikeinen, joka tarkoittaa, että sovellukseen on mahdollista tehdä taustalla ajettavia toimintoja ja tehtäviä säikeiden avulla. (Tutorialspoint n.d.a.)

Ajettaessa Javalla ohjelmoitu sovellus, laitteelle asennettu Java-alusta kääntää koodin laiteyhteensopivaksi tavukoodiksi. Tämä tarkoittaa sitä, että Javalla ohjelmoitu sovellus on helposti siirrettävissä ja ajettavissa eri laitteilla ja käyttöjärjestelmissä, kunhan sovelluksessa ei ole käytetty muiden kielien kirjastoja ja laitteella on Java-tuki asennettuna. (Oracle n.d.b.)

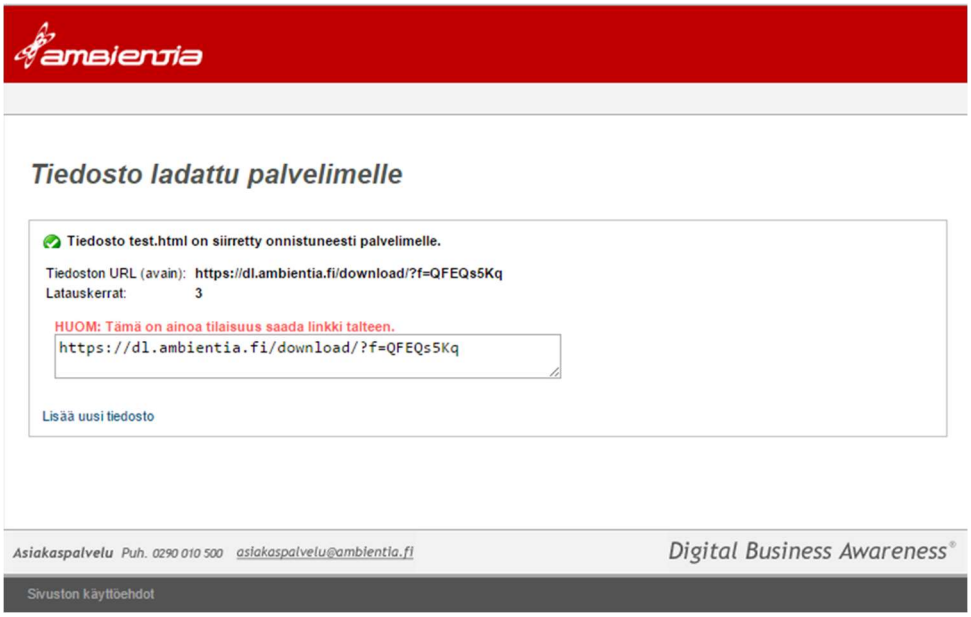
2.2 Sovelluksen toiminta ja käyttötapaukset

Sovellus on toiminnaltaan ja käyttötapauksiltaan yksinkertainen. Etusivuna aukeaa käyttöliittymä, jonka avulla tiedosto lähetetään palvelimelle (Kuva 1). Käyttäjä painaa Choose File -painiketta, jonka jälkeen hän voi valita yhden enintään kaksi gigatavua suuren tiedoston. Tämän jälkeen asetetaan tiedoston latauskerrat eli kuinka monta kertaa palvelimelle tallennettu tiedosto voidaan ladata. Lopuksi Lähetä-painikkeella aloitetaan tiedoston lähetys.



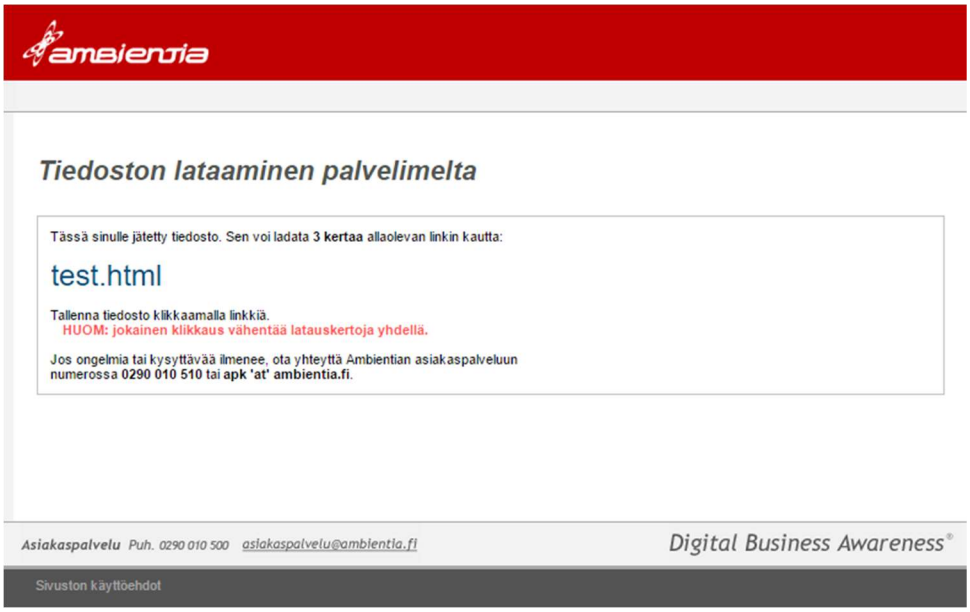
Kuva 1. Sovelluksen etusivu ja tiedoston lähetys palvelimelle

Kun tiedosto on onnistuneesti lähetetty palvelimelle, käyttäjä ohjataan uudelle sivulle (Kuva 2). Sivulla kerrotaan, minkä niminen tiedosto on juuri ladattu palvelimelle ja kuinka monta kertaa se on mahdollista ladata. Tältä sivulta löytyy myös sovelluksen generoima linkki. Linkki ohjaa sivulle, jolta voi ladata palvelimelle lähetetyn tiedoston. Generoituneeseen linkkiin ei pääse käsiksi enää jälkikäteen, joten se täytyy ottaa tässä vaiheessa talteen, jos sen haluaa jakaa eteenpäin.



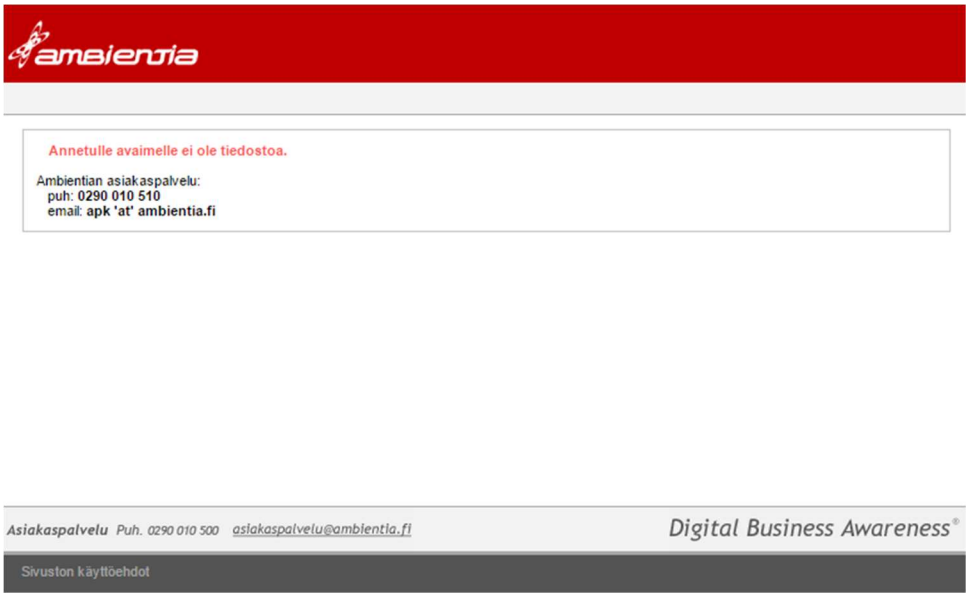
Kuva 2. Sovellus ilmoittaa onnistuneen latauksen tiedot ja tarjoaa generoidun linkin

Generoitunut linkki ei vielä lataa tiedostoa palvelimelta vaan se ohjaa käyttäjän kuvan 3 mukaiselle uudelle sivulle. Tältä sivulta löytyy tieto, kuinka monta kertaa tiedosto on vielä mahdollista ladata sekä itse latauslinkki. Tiedoston latauskerrat päivittyvät reaaliajassa, jotta lataaja näkee, montako kertaa tiedoston voi vielä ladata.



Kuva 3. Tiedoston lataussivu

Kuvassa 4 on lopputilanne, kun tiedoston latauskerrat loppuvat. Linkkiä painaessa lataaja ohjataan sivulle, jolla kerrotaan, että tiedosto ei ole enää ladattavissa.



Kuva 4 Tiedosto ei ole enää ladattavissa

3 JAVASCRIPT-PILVIPALVELUSOVELLUS

Alkuperäistä sovellusta ei ollut tarkoitus muuttaa vaan toteuttaa se uudelleen hyödyntämällä uusia tekniikoita. Tässä luvussa käsitellään uuden sovelluksen tietoperusta. Tietoperustaan kuuluu sovelluksen toteutuksen ja toiminnan kannalta oleellisimmat tekniikat: OpenShift, Node.js ja AngularJS. Luvussa käydään läpi edellä mainittujen tekniikoiden pääkonsepteja ja niiden toimintaa.

3.1 AngularJS

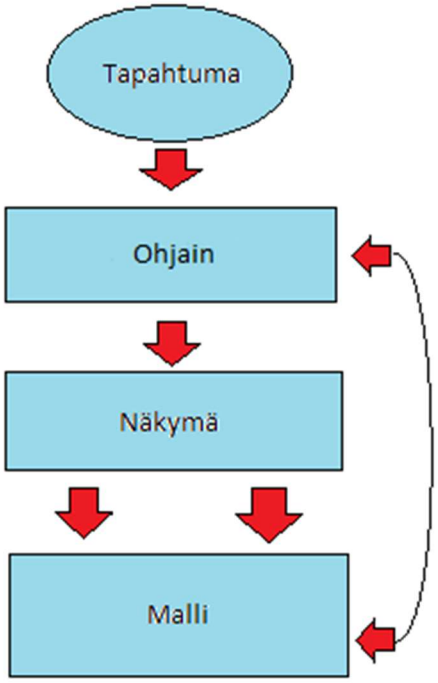
AngularJS on Googlen työntekijöiden kehittämä ja ylläpitämä JavaScript-sovelluskehys (framework). Sen on alun perin luonut kaksi kehittäjää, Misko Hevery ja Adam Abrons vuonna 2009. Tuolloin se kantoi nimeä GetAngular. AngularJS:n potentiaali huomattiin Googlella toisen projektin yhteydessä, jossa projektin koodin määrää saatiin huomattavasti vähennettyä sen avulla. Abrons jäi myöhemmin kehityksestä pois, mutta Heverey jatkoi sitä esimiehensä Brad Greenin kanssa. He aloittivat projektin AngularJS-nimellä, perustivat työryhmän ja aloittivat sen ylläpidon Googlella. (Austin 2014.)

AngularJS on rakenteellinen, MVC-malliin perustuva sovelluskehys dynaamisten verkkosovellusten toteuttamiseen. AngularJS loistaa varsinkin yksisivuisten verkkosovellusten toteuttamisessa. Kaikki sen ohjelmakoodi ajetaan selaimessa, joten se ei ole riippuvainen palvelinpuolesta. (Shirastava 2014.)

AngularJS:llä tehty sovellus rakentuu moduuleista, jotka ovat riippuvaisia toisistaan. AngularJS:n tiedon sitominen (data binding) ja riippuvuusinjektio (dependency injection) vähentää sovelluksen koodin määrää huomattavasti. AngularJS laajentaa HTML-syntaksia direktiiveillä, joiden avulla DOM:in manipulointi dynaamisesti on helpompaa ja nopeampaa, kuin esimerkiksi AJAX:a käyttämällä. (Bégaudeau 2014.)

3.1.1 MVC-arkkitehtuuri

MVC tulee sanoista Model, View ja Controller. Se on sovellusrakenne, jota käytetään websovellusten kehityksessä ja jota myös AngularJS hyödyntää. Sen pääajatus on, että sovellus päivittää itseään reaaliaikaisesti ilman erillisiä latauksia tai sivupäivityksiä. MVC mahdollistaa sovelluksen toiminnallisen logiikan eristämisen käyttöliittymästä. Ohjain (Controller) vastaanottaa ja käsittelee kaikki pyynnöt ja datan. Tämän jälkeen ohjain käsittelee näkymän (View) tarvitseman datan mallin (Model) kanssa ja lähettää sen näkymälle. Näkymä luo vastaanottamastaan datasta lopullisen näkymän, joka näytetään käyttäjälle selaimessa. (Tutorialspoint n.d.b.) Kuvassa 5 on mallinnus MVC-rakenteesta ja sen toiminnasta.



Kuva 5. MVC-rakenne ja toimintapolku (Tutorialspoint n.d.b.)

Malli on vastuussa sovelluksen datan hallitsemista. Se päivittää itseään näkymässä tapahtuvien muutosten ja ohjaimen toimintojen mukaan. Näkymä esittää datan käyttäjälle sen käyttämässä formaatissa. Näkymä kuuntelee ohjainta ja esittää dataa sitä mukaan, kun se vastaanottaa sitä. Ohjain kuuntelee käyttäjän toimintaa ja mallia. Ohjain tekee muutoksia malliin sitä mukaa, kun käyttäjä on vuorovaikutuksessa käyttöliittymän kanssa. Tämän jälkeen ohjain vastaanottaa päivittyneen datan mallista ja välittää sen näkymälle. (Tutorialspoint n.d.b.)

3.1.2 Dokumenttipohja, tiedon sitominen ja direktiivit

AngularJS:ssä dokumenttipohja (template) kirjoitetaan HTML-kielellä, johon voidaan lisätä AngularJS:n omia elementtejä ja attribuutteja. AngularJS yhdistää dokumenttipohjan mallista saadun datan kanssa, jonka jälkeen ohjain muodostaa selaimeen dynaamisen näkymän. Dokumenttipohjaa voidaan täydentää direktiiveillä ja lausekkeilla. (Google 2016a.)

AngularJS:n direktiivit ovat DOM-elementteihin lisättäviä merkintöjä. Direktiivien avulla AngularJS:n oma HTML-kääntäjä pystyy liittämään DOM-elementteihin toimintoja tai muuttamaan itse elementtiä tai sen alielementtejä. (Google 2016b.)

Lausekkeet (expressions) ovat lyhyitä koodin pätkiä, joita käytetään pääasiassa interpoloinnin yhteydessä, mutta niitä voidaan käyttää suoraan myös direktiiveissä. Lausekkeet ja niiden interpolointi mahdollistavat HTML:n ja näkymän dynaamisen ja manipuloinnin sekä skoopin muuttujien esittämisen ja lisäämisen näkymään. (Google 2016f.)

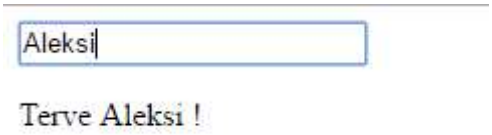
AngularJS-websovelluksissa tiedon sitominen vastaa datan automaattisesta synkronoinnista mallin ja näkymän välillä, kun sovellus on käynnissä. Kun selain kääntää AngularJS-dokumenttipohjan, se luo reaaliaikaisen näkymän. Tiedon sitominen toimii MVC-mallin mukaan eli muutokset näkymässä päivittyvät reaaliaikaisesti malliin ja päinvastoin ilman erillisiä sivun latauksia. Malli toimii ainoana tiedonlähteenä, jonka kautta data kulkee. (Google 2016c.)

Kuva 6 demonstroi tiedon sitomisen ja direktiivien käyttöä yksinkertaisella tasolla. AngularJS otetaan käyttöön script-tagien avulla aivan, kuten muutkin JavaScript-tiedostot. Ensimmäinen direktiivi, joka on AngularJS:n toiminnan kannalta pakollinen, on body-tagin sisään lisätty ng-app. Tämä direktiivi merkitsee, että AngularJS-sovellus alkaa tästä. Direktiivi voisi olla myös heti html-tagin sisällä. Heittomerkkien sisään voidaan lisätä sille nimi, mutta sitä ei tässä tapauksessa tarvita. Seuraava direktiivi on ng-model, joka lisää malliin inputName nimisen muuttujan. Tämä muuttuja tulostetaan sivulle lausekkeen avulla lisäämällä muuttujan nimi kaksinkertaisten aaltosulkeiden sisään. Kuvissa 7 ja 8 näkyy koodin toiminta selaimessa. Käyttäjän syöttäessä tekstikenttään jotakin, se tulostuu sivulle reaaliaikaisesti.

```
<!DOCTYPE html>
<html>
    <meta charset="UTF-8">
    <head>
        <script src="angular.min.js"></script>
    </head>
    <body ng-app="">
        <div>
            <input placeholder="Etunimesi" type="text" ng-model="inputName" />
            <p>Terve {{inputName}} !</p>
        </div>
    </body>
</html>
```

Kuva 6. Tiedon sitomisen ja direktiivien esimerkki koodi

Kuva 7. Alkutilanne selaimessa



Kuva 8. Muuttujan tulostuminen reaaliaikaisesti

3.1.3 Moduulit ja riippuvuusinjektio

Moduulit toimivat säiliöinä sovelluksen eri osille, kuten ohjaimille, palveluille, direktiiveille. Moduuli kokoaa ohjelman eri komponentit yhteen. Muilla sovelluksilla on yleensä päämetodi, joka käynnistää sovelluksen. AngularJS-sovelluksilla ei ole päämetodia, vaan moduuleissa määritetään, miten sovellus otetaan käyttöön. Moduulien avulla koodia on helpompi lukea ja sovellus voidaan jakaa uudelleenkäytettäviin osiin. (Google 2016g.)

Riippuvuusinjektio mahdollistaa sovelluksen jakamisen eri osiin. Sen avulla sovellus pääsee käsiksi sille määritettyihin komponentteihin, jotka on ohjelmoitu irrallisina. Sovellukselle voidaan asettaa riippuvuusinjektion avulla ohjaimia, direktiivejä ja palveluita. Käynnistyessä sovellus tarkistaa sen riippuvuudet ja lataa sille määritetyt komponentit. (Google 2016h.)

3.1.4 Ohjain ja skooppi

AngularJS:ssä ohjain määritetään JavaScript-funktiolla. Kun ohjain liitetään DOM:iin, AngularJS luo uuden ohjain -objektin sille määritetyn nimen mukaan. Ohjainta luotaessa on mahdollista injektoida skooppi (scope)-objekti sen käyttöön. Ohjaimen avulla voidaan luoda skooppi-objektiin muuttujia ja funktioita. Ohjaimen kuuluu vastata vain sovelluksen bisneslogiikasta eli sovelluksen toiminnan kannalta oleellisimmista asioista. (Google 2016d.)

Skooppi on objekti, joka viittaa sovelluksen malliin. Se toimii lausekkeiden suoritusympäristönä. Skooppi seuraa lausekkeita ja tapahtumia mallin ja näkymän välillä. Skooppi yhdistää sovelluksen ohjaimen ja näkymän sekä toimii datan välittäjänä niiden välillä. Kun dokumenttipohja muodostetaan selaimeen, direktiivit lisäävät seurantalausekkeita skooppiin. Nämä lausekkeet ilmoittavat direktiiveille muutoksista, jonka avulla ne voivat esittää päivittyneen datan selaimessa reaaliaikaisesti. (Google 2016e.)

Kuvassa 9 demonstroidaan moduulin ja ohjaimen käyttöä AngularJS-sovelluksessa. Sovelluksen ohjaimen koodi on kirjoitettu script-tagin sisään. Se alkaa moduulin ja ohjaimen määrittämisellä. Moduuli tarkoittaa tässä tapauksessa myös itse sovellusta ja sille annetaan nimi myApp. Ohjaimelle annetaan nimi myCtrl. Tämän jälkeen ohjaimelle asetetaan skooppi riippuvaisuudeksi, jotta sovellus osaa ottaa sen käyttöön sovelluksen käynnistyessä. Näiden määritysten jälkeen luodaan oma funktio ohjainta varten, jolle annetaan parametrina riippuvaisuudeksi asetettu skooppi-objekti. Funktion sisällä skooppin asetetaan kaksi muuttujaa ja yksi funktio.

Body-tagin sisään on lisätty ng-app-direktiivi, jolle on asetettu sama nimi kuin moduulille, jotta sovellus toimii. Tämän jälkeen div-elementille on annettu ng-controller-direktiivi, jolle annetaan luodun ohjaimen nimi. Ohjain toimii vain tämän div-elementin sisällä. Sovellus toimii siten, että käyttäjän syöte tulostuu sivulle napin painalluksella. Skooppiin lisättyä funktiota kutsutaan ng-click-direktiivillä button-tagin sisällä. Funktio asettaa skoopin name-muuttujan arvon inputName-muuttujan mukaiseksi. Käyttäjän syöte

tulostuu sivulle lausekkeen avulla, jonka sisään on ketjutettu skoopin greet- ja name-muuttuja. Kuvissa 10 ja 11 on sovelluksen toiminta selaimessa.

```
<!DOCTYPE html>
<html>
    <meta charset="UTF-8">
    <head>
        <script src="angular.min.js"></script>
        <script>
            angular
                .module('myApp', [])
                .controller('myCtrl', myCtrl);

            myCtrl.$inject = ['$scope'];

            function myCtrl ($scope) {

                $scope.greet = "Terve";
                $scope.name = "";

                $scope.setName = function () {
                    $scope.name = $scope.inputName;
                }
            }
        </script>
    </head>
    <body ng-app="myApp">
        <div ng-controller="myCtrl">
            <input placeholder="Etunimesi" type="text" ng-model="inputName" />
            <button ng-click="setName()">Aseta nimi</button>
            <p>{{greet + " " + name}} !</p>
        </div>
    </body>
</html>

```

Kuva 9. Ohjaimen käytön esimerkkikoodi

Kuva 10. Alkutilanne selaimessa

Kuva 11. Reaaliaikaisesti tulostunut nimi

3.2 Node.js

Node.js on palvelintasolle tarkoitettu JavaScript-alusta, jonka on luonut Ryan Dahl vuonna 2009. Node.js on rakennettu Chromen V8 JavaScript-moottorin päälle. Node.js:n avulla on mahdollista luoda kevyitä, skaalautuvia, nopeita, tapahtumapainotteisia ja järjestelmää tukkimattomia sovelluksia. (Martínez 2013.)

Node.js:n keveyden ja pienien teknisten vaatimusten takia se on parhaimmillaan pilvialustoilla. Muihin perinteisiin sovelluskehyksiin verrattuna Node.js on erilainen, koska se on hyvin tapahtumapainotteinen. Sovellukset ohjelmoidaan lähtökohtaisesti asynkronisesti callback-mekanismia hyödyntäen. Tämän etuna on esimerkiksi se, että yksittäisiin HTTP-kutsuihin voidaan liittää useita asynkronisia palvelukutsuja tukkimatta järjestelmää. Tämä mahdollistaa reaaliaikaisten sovellusten tekemisen. Asynkronisen callback-mekanismin käyttö on myös yksi merkittävimmistä haasteista. Sovelluksen kasvaessa, callback-kutsuista muodostuu hyvin äkkiä todella pitkiä ja monimutkaisia ketjuja, joita on vaikea lukea ja ymmärtää. (Salonen 2012.)

3.2.1 Asynkroninen käyttöjärjestelmä ja toiminta

Node.js on palvelintason sovelluskehys, joten yksi sen päätehtävistä on käsitellä ja vastaanottaa selaimelta tulevia pyyntöjä (request). Perinteiset pyyntöjä vastaanottavat ja välittävät käyttöjärjestelmät käsittelevät niitä lineaarisessa järjestyksessä eli seuraava kutsu käsitellään vasta, kun edellinen on käsitelty. Tämä tarkoittaa sitä, että selain joutuu odottamaan sen aikaa, kun järjestelmä käsittelee pyynnön ja vastaa siihen. (Singh 2015.)

Kuvassa 12 demonstroidaan perinteisen järjestelmän lineaarista toimintaa. Koodi demonstroi tarjoilijan toimintaa. Tarjoilija ottaa vastaan ensimmäisen tilauksen, odottaa, kunnes tilaus on valmis ja tarjoilee sen. Vasta tämän jälkeen tarjoilija ottaa vastaan seuraavan tilauksen.

```
14  // Ota ensimmäinen tilaus
15  var order = "Coke";
16
17  // Odota, että tilaus valmis
18
19  // Tarjoile ensimmäinen tilaus
20  serveOrder(order);
21
22  // Ota toinen tilaus
23  var order2 = "Coffee";
24
25  // Odota, että tilaus valmis
26
27  // Tarjoile toinen tilaus
28  serveOrder(order2);
```

Kuva 12. Perinteisen järjestelmän lineaarinen toiminta (Singh 2015.)

Node.js käsittelee kutsut siten, että ne kutsut joiden käsittely kestää pidempään asetetaan tapahtuma jonoon (event loop). Tapahtuma jonossa niiden käsittely jatkuu edelleen, mutta nyt sovellus siirtyy käsittelemään seuraavaa kutsua tai tehtävää, jäämättä odottamaan edellisen valmistumista. Tapahtuma jonoon siirretty kutsu tai tehtävä esitetään tai toteutetaan heti, kun se on käsitelty. Tätä toimintaa kutsutaan asynkroniseksi, jossa tehtävät ja kutsut käsitellään niin sanotusti taustalla tukkimatta järjestelmää. (Singh 2015.)

Kuva 13 demonstroi Node.js:n tehtävien käsittelyä ja kutsujen käsittelyä. Esimerkkinä käytetään jälleen tarjoilijan toimintaa. Tällä kertaa sen sijaan, että tarjoilija jää odottamaan tilauksen valmistumista, hän jatkaa eteenpäin ottamaan lisää tilauksia ja tarjoilee tilaukset, kun ne valmistuvat.

```
14  // Ota tilaus ja ota vastaan seuraava..
15  takeOrder("Coffee", function (order) {
16      // Tarjoile tilaus kun valmis
17      return serveOrder(order);
18  });
19
20  // Ota tilaus ja ota vastaan seuraava..
21  takeOrder("Coke", function (order2) {
22      // Tarjoile tilaus kun valmis
23      return serveOrder(order2);
24  });
```

Kuva 13. Node.js tehtävien käsittely (Singh 2015.)

3.2.2 Callback

JavaScript-funktiot ovat ensimmäisen luokan olioita, mikä tarkoittaa sitä, että niitä voidaan hyödyntää aivan, kuten muitakin olioita. Funktio voidaan asettaa muuttujaan, metodin parametriksi, olion arvoksi sekä se voidaan palauttaa funktiosta. Callbackit ovat JavaScriptin anonyymejä funktioita, joita voidaan asettaa toisen funktion parametreiksi. Callback-funktio voidaan suorittaa tai palauttaa funktiosta myöhemmin suoritettavaksi. Sitä voidaan käyttää määrittämällä se toisen funktion parametriksi. Callback-funktion suoriutuminen riippuu täysin funktion toiminnasta, jossa sitä käytetään. Funktio ajetaan vasta, kun sen niin sanottu isäntäfunktio on suoriutunut. Tästä syystä ei voida tarkalleen tietää, milloin callback-funktio suoritetaan. Callback-funktiot luovat pohjan Node.js:n järjestelmää tukkimattomalle ja asynkroniselle toiminnalle. (Singh 2015.)

Kuvassa 14 on kirjoitettuna yksinkertainen demonstraatio callback-funktion käytöstä. Kuvassa annetaan setTimeout-funktion parametrina anonyymi funktio, joka tulostaa konsoliin sanan world. Anonyymi funktio ei tiedä milloin sen pitäisi suoriutua. Tässä tapauksessa funktion suoriutuminen riippuu täysin funktiosta, jonka sisällä se on. SetTimeout-funktio on asetettu odottamaan kaksi sekuntia, jonka jälkeen sen parametriksi asetettu anonyymi funktio suoriutuu. Vaikka setTimeout-funktio on ennen hello-sanan tulostusta, hello-sana tulostuu ensin. Tämä johtuu siitä, että järjestelmä asettaa setTimeout-funktion suoriutumaan tapahtuma jonoon ja jatkaa eteenpäin. Kun kaksi sekuntia on kulunut, se tulostaa world-sanan.

```
14  setTimeout(function() {
15      console.log("world");
16  }, 2000)
17
18  console.log("hello");
19
20  //tuloste
21  hello
22  world
```

Kuva 14. Yksinkertaisen callback-funktion käyttäminen (Singh 2015.)

3.2.3 Moduulit

Node.js:ssä moduulit ovat JavaScript-tiedostoja, jotka sisältävät koodia eri toimintoihin ja tarkoituksiin. Moduuleita käytetään koodin jakamisessa osiin ja täten sen selkeyttämisessä. Moduulit helpottavat koodin lukua ja ehkäisevät hyvin pitkien kooditiedostojen muodostumista sekä saman koodin toistumista. (Singh 2015.)

Moduulit voidaan jakaa kahteen eri kategoriaan: ydinmoduuleihin ja kehittäjän omiin moduuleihin. Kehittäjän itse asentamat paketit sisältyvät myös ydinmoduuleihin ja ne otetaan käyttöön samalla tavalla. Ydin moduulit tulevat Node.js kirjaston mukana ja niiden tarkoitus on helpottaa kehittäjän työskentelyä tarjoamalla uudelleenkäytettävää ja toistuvaa koodia. Näin kehittäjä säästyy koodin jatkuvalta uudelleen kirjoittamiselta. (Singh 2015.)

Kehittäjän omat moduulit ovat nimensä mukaisesti kehittäjän itse luomia, johonkin tiettyyn tarkoitukseen. Omien moduulien kirjoittaminen voi tulla tarpeeseen, jos ydinmoduulit eivät täytä kehittäjän tarpeita. (Singh 2015.)

Moduuleita voidaan hyödyntää vaatimalla niitä koodissa. Kuvassa 15 esitetään, miten moduuli voidaan ottaa käyttöön. Ydinmoduuli otetaan käyttöön vaatimalla moduulia sen nimellä. NPM:n avulla asennetut paketit otetaan käyttöön myös samalla tavalla, koska ne sisältyvät ydin moduuleihin. Käyttäjän oma moduuli otetaan käyttöön kirjoittamalla moduulin nimen tilalle polku siihen tiedostoon, johon moduuli on kirjoitettu.

```
14  // Ydin moduulin käyttöönotto
15  var http = require('http');
16
17  // Kehittäjän oman moduulin käyttöönotto
18  var omaModuuli = require('./projekti/moduulit/omaModuuli.js');
```

Kuva 15. Moduulien käyttöönotto (Singh 2015.)

Omaan moduuliin kirjoitettuja funktioita, muuttujia tai muita objekteja voidaan käyttää muissa tiedostoissa, jos moduuli jakaa ne. Kuvissa 16 ja 17 on demonstraatio, kuinka moduuli voi jakaa funktion ja kuinka sitä voidaan käyttää toisessa tiedostossa. Kuvassa 16 on kirjoitettuna funktio, joka tahdotaan jakaa. Funktion jakaminen määritetään exports-objektin avulla. Tämä määrittää, että moduuli jakaa kirjoitetun funktion sille määritetyllä

nimellä. Jaettua funktiota voidaan käyttää toisessa tiedostossa ensin vaatimalla kirjoitettua moduulia ja tämän jälkeen kutsumalla funktiota (Kuva 17).

```
23  var moduuliFunktio = function () {
24      console.log(Olen moduulin X funktio)
25  }
26
27  exports.moduuliFunktio = moduuliFunktio;
```

Kuva 16. Omaan moduuliin kirjoitettu funktio

```
18  var omaModuuli = require("./oman/moduulin/polku");
19
20  omaModuuli.moduuliFunktio();
```

Kuva 17. Moduulin funktion kutsuminen

3.2.4 NPM

NPM on Node.js:n oma pakettienhallintajärjestelmä, jonka avulla voi asentaa paketteja ja koodia sekä hallita sovelluksen riippuvaisuuksia. Node.js:n päätarkoitus on, että sen ydin on kevyt ja suurin osa sovelluksen toiminnoista toteutetaan ulkoisten pakettien avulla. Tästä syystä NPM kulkee käsi kädessä Node.js:n kanssa. NPM:n avulla paketteja voidaan hallita moduulikohtaisesti ja ne asentuvat automaattisesti, mikäli kyseinen paketti on asetettu sovelluksen riippuvaisuudeksi package.json-tiedostoon. Riippuvaisuudet asentuvat, kun sovellus käynnistetään. (Salonen 2012.)

Paketit voidaan asentaa globaalisti, joka toimii käyttöjärjestelmäkohtaisesti tai projektikohtaisesti. Paketti voidaan asentaa projektikohtaisesti ajamalla projektin juuressa komento: npm install ’paketin nimi’. Lisäämällä komennon perään -save, paketti asetetaan sovelluksen riippuvaisuudeksi. (Salonen 2012.) Kuvassa 18 esitetään sovelluksen yksikertainen package.json-tiedoston rakenne. Kuvassa näkyy myös asennuksen jälkeen riippuvuudeksi listattu paketti.

app.js | package.json

```
1  {
2    "name": "app",
3    "dependencies": {
4      "express": "^4.14.0"
5    }
6  }
7
```

Kuva 18. Package.json-tiedsto ja sen rakenne

3.2.5 Express

Express on Node.js-sovelluskehys, joka tarjoaa laajan valikoiman toimintoja ja ominaisuuksia verkkosovellusten kehitykseen. Expressin etu on, että sen kirjasto sisältää niin paljon valmiita toimintoja sovelluksen luomiseen, joiden ohjelmointi Node.js:n oman kirjaston avulla vaatisi huomattavasti enemmän työtä. (Strongloop 2016a.)

Expressissä reitityksellä viitataan siihen, kuinka sovellus vastaa käyttäjältä vastaanotettuun pyyntöön, joka kohdentuu johonkin sovelluksen URL-osoitteeseen. Jokaiselle reitille voidaan määrittää oma funktio, joka ajetaan, kun polkuun saapuu pyyntö. Jokaisella reitillä voi olla yksi tai useampi käsittelijä ja ne voidaan erotella HTTP-metodien mukaan. Reittien käsittelyä varten voidaan kirjoittaa myös erillinen moduuli ja erottaa reitit pääsovelluksesta. (Strongloop 2016b.)

Express hyödyntää sovellukseen saapuvien pyyntöjen käsittelyssä request- ja response-objekteja. Request-objekti edustaa HTTP-pyyntöä, joka sisältää sille tyypillisiä tietoja, kuten pyynnön mukana tulevia parametreja ja dataa, HTTP-headereita ja muita tietoja pyytävästä osapuolesta. Response-objekti edustaa taas vastausta, jonka Express-sovellus lähettää, kun se vastaanottaa pyynnön. (Strongloop 2016c.)

Kuvassa 19 on demonstraatio yksinkertaisesta Express-sovelluksesta. App.js-tiedostoon ollaan kirjoitettu koodi, joka käynnistää palvelinsovelluksen, joka alkaa kuunnella porttiin 3000 saapuvaa liikennettä. Kun sovellus on käynnistynyt ja yhdistänyt porttiin 3000, se lähettää lokiviestin sen merkiksi. Sovellus vastaa ”Hello World!” sen juureen saapuviin pyyntöihin ja 404-koodilla sellaisiin pyyntöihin, joille ei ole reittiä määritettynä.

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (request, response) {
5    response.send('Hello World!');
6  });
7
8  app.listen(3000, function () {
9    console.log('Esimerkki sovellus kuuntelee porttia 3000!');
10 });
```

Kuva 19. Yksinkertaisen Express-sovelluksen koodi

Kuvassa 20 on luotuna erillinen moduuli routes.js-tiedostoon, joka määrittelee reitit ja niiden tavat käsitellä vastaanotettuja pyyntöjä. Moduulin tekoon on käytetty Expressin Router-luokkaa. Kuvassa 21 moduuli otetaan käyttöön pääsovelluksessa app.use-funktiolla. Ensin määritetään URL-osoite, jonka perään reitit liitetään. Tämän jälkeen annetaan muuttuja, jossa on vaadittu reittimoduuli. Sovellus voi vastata nyt /tests/test- ja /tests/test2-osoitteisiin saapuviin pyyntöihin kuvan 20 mukaisilla tavoilla.

app.js routes.js

```
1  var express = require('express');
2  var router = express.Router();
3
4  router.route('/test1')
5
6      .get(function (request, response) {
7
8          response.send('Moi ! Tämä on /test1');
9
10     });
11
12 router.route('/test2')
13
14     .get(function (request, response) {
15
16         response.send('Moi ! Tämä on /test2');
17
18     });
19
20 module.exports = router;
```

Kuva 20. Reititysmoduulin koodi

```
12  var routes = require('./routes');
13
14  ...
15
16  app.use('/tests', routes);
```

Kuva 21. Reittien käyttöönotto reititysmoduulista

3.3 OpenShift Online

OpenShift Online on Red Hatin PaaS-pilvipalvelualusta. PaaS on palvelumuoto, joka on pääasiassa suunnattu sovelluskehittäjille ja ohjelmoijille. Palveluun kuuluu yleensä, että kehittäjä voi valita palvelun tarjoamista osista, joiden avulla hän haluaa kehittää sovellusta. OpenShiftin päällä voidaan kehittää, testata ja ylläpitää sovelluksia Red Hatin pilviympäristössä. Kehittäjän ei tarvitse huolehtia oman palvelimen pysyttämisestä ja ylläpitämisestä, joten hän voi keskittyä täysin sovelluksen kehittämiseen (OpenShift 2016a.) OpenShift tarjoaa useita eri ohjelmointikieliä, palvelin, sovelluskehys, tietokanta ja kehitystyökalu vaihtoehtoja. Sovellukseen valittavista sovelluskehyksistä, tietokannoista ja teknologioista puhutaan nimillä koneisto (gear) ja kasetti (cartridge). (OpenShift 2016b.)

Koneisto on palvelinsäiliö, jolla on tarvittavat resurssit eri sovellusten ajamiseen ja ne toimivat OpenShiftin pilvessä. Koneiston sisään asennetaan kasetteja, joilla voidaan luoda sovellus. Koneistoista on tarjolla kolme eri tyyppiä: pieni, keskikokoinen ja suuri. Jokainen koko tarjoaa oletuksena yhden gigatavun levytilaa ja niiden muistin määrä vaihtelee 512 megatavusta kahteen gigatavuun. (OpenShift 2016a.)

Kasetit ovat lisäosia koneistoille, jotka voivat sisältää sovelluskehyksen tai komponentteja, joiden avulla voidaan luoda sovellus ja ajaa sitä. Kasetteja on kahdenlaisia: erillisiä (standalone) ja sulautettuja (embedded). Erilliset kasetit sisältävät ohjelmointikieliä tai palvelinsovelluksia, joiden avulla on tarkoitus jakaa verkkosisältöä, kuten Node.js. Sulautetun kasetin avulla on tarkoitus parantaa sovelluksen toiminnallisuutta, mutta sen avulla pelkästään ei voida luoda sovellusta. Sulautettu kasetti voi olla esimerkiksi tietokanta. (OpenShift 2016a.)

OpenShiftin avulla voidaan hoitaa sovelluksen horisontaali skaalaaminen. Se mahdollistaa sovelluksen reagoimisen verkkoliikenteessä tapahtuviin muutoksiin. Tämä tarkoittaa, että sovellus voi monistamalla itseään jakaa sen resursseja, kun sovellukseen kohdentuva kuorma lisääntyy. OpenShift infrastruktuuri monitoroi saapuvaa liikennettä ja automaattisesti monistaa sovellusta, kun siihen alkaa kohdentua paljon liikennettä. Liikennekuorman käsittely jakaantuu monistuneen sovelluksen kesken eikä sovellus ylikuormitu niin helposti. (OpenShift 2016a.)

OpenShift-alustan kanssa voidaan olla vuorovaikutuksessa, joko OpenShift-verkkokäyttöliittymän tai RHC-komentotyökalun avulla. Näiden työkalujen käyttäminen on pakollista vain luodessa uutta sovellusta tai hallitessa sen infrastruktuuria, kuten liittäessä uutta kasettia sovellukseen. OpenShiftin-alustan kautta sovellusta voidaan myös monitoroida, ajaa ja hienosäätää. Muuten sovellusta työstetään pääasiassa Gitin ja SSH:n avulla. (OpenShift 2016b.)

Sovellusta voidaan kehittää omilla työkaluilla ja versionhallintaohjelmalla. OpenShift hoitaa sovelluksen uuden version käyttöönoton ja käynnistämisen sekä sen toimittamisen käyttäjien saataville. OpenShiftistä on saatavilla eri hintatasoja ilmaisesta versiosta lähtien. Hintatasojen mukaan vaihtelee mahdollisten luotavien sovellusten määrä, niiden teho ja muut lisäominaisuudet. (OpenShift 2016b.)

4 SOVELLUKSEN TOTEUTUS

Tässä luvussa käydään läpi tehdyn työn käytännönosuus. Toimeksiantona oli toteuttaa uusi versio toimeksiantajan alkuperäisestä sovelluksesta AngularJS:n ja Node.js:n avulla OpenShift Online -alustalle. Luvussa tullaan käymään läpi vain uuden sovelluksen pääominaisuuksien ja toimintojen toteutus. Osuuden tarkoitus on toimia tukimateriaalina koodia läpikäydessä. Luvussa ei tulla käymään läpi sovelluksen toiminnan kannalta vähemmän oleellisia asioita, kuten ulkoasua. Toimintojen kannalta oleellisten tiedostojen rakennetta ei myöskään välttämättä käydä kokonaan läpi, vaan ainoastaan ne osat, jotka ovat sovelluksen päätoimintojen kannalta oleellisia.

Sovellusta kehitettiin suurimmaksi osaksi paikallisesti omalla tietokoneella. OpenShift otettiin käyttöön vasta myöhemmin, koska se ei ollut saatavilla kuin vasta myöhemmin projektin edettyä. Sovelluksen siirtäminen OpenShiftiin paikallisen kehityksen jälkeen pakotti odotetusti tekemään pieniä muutoksia koodissa, mutta se ei tuottanut suuria ongelmia. Sovelluksen paikallisesti kehittäminen oli huomattavasti nopeampaa kuin OpenShift-alustalla. Uuden koodin siirtäminen OpenShiftiin saattoi välillä viedä useita minuutteja, joten koodin toiminnan testaaminen oli hyvin hidasta. Kehitys toteutettiin Atom-ohjelmointieditorilla ja versionhallinta SmartGit-ohjelmalla.

Sovellusta kehitettiin uusimmalla ja vakaimmalla AngularJS-versiolla 1.5.7. Node.js-puolen ohjelmoimiseen käytettiin Express-sovelluskehystä. Paikallisesti sovellusta kehittäessä, Node.js:stä käytettiin vakainta ja suositelluinta versiota 4.4.7, mutta kun kehitys siirtyi OpenShift-alustalle, oli valittava eri versio. OpenShift tarjoaa Node.js:stä vain versiota 0.x.x tai kustomoitua Node.js-lisäosaa, joka päivittyy aina uusimpaan saatavilla olevaan versioon. Ei ollut varmuutta, että toimisiko sovellus 0.x.x-version päällä, koska sovellusta oltiin kehitetty uudemmalla versiolla. Tästä syystä valittiin kustomoitu Node.js lisäosa varmuuden vuoksi. Alustasta määritettiin automaattisesti skaalautuva.

Tietokannaksi valittiin PostgreSQL. Paikallisesti kehittäessä, tietokannasta käytettiin versiota 9.4, mutta siihen tuli muutos, kun sovelluksen kehitys siirtyi OpenShiftiin, koska OpenShift tarjosi vain versiota 9.2. Tästä kehittyi myös pieni hetkellinen ongelma. Opin kannalta, tietokannassa käytettiin alun perin jsonb-tietotyyppiä, mutta PostgreSQL:n 9.2-versio ei tukenut tätä, joten tietokannan skeema jouduttiin laittamaan uusiksi.

4.1 Tiedoston lähetys palvelimelle

Toimeksiantajan sovelluksen alkuperäisessä versiossa tiedoston lähettäminen on tehty hyvin yksinkertaiseksi. Sama yksinkertaisuus haluttiin säilyttää myös uudessa versiossa, mutta muutaman uuden lisäominaisuuden kera. Uutena ominaisuutena lisättiin:

- drag and drop tiedoston valitseminen
- vanhentumispäivämäärän valinta
- lähettäjän sähköpostin syöttö

- latauslinkin lähetys sähköpostilla.

Tiedoston lähetys -sivua varten kirjoitettiin ohjain erilliseen uploadFilePage.js-tiedostoon. Tiedostossa määritetään sovelluksen ja ohjaimen riippuvuudet. Kuvassa 22 on sovelluksen ja ohjaimen määritykset. Sovelluksen nimi on app ja sille määritettiin hakasulkeiden sisällä sen riippuvuudet. Riippuvuuksien toimintaa käydään tarkemmin läpi myöhemmin tässä luvussa. Ohjaimen nimeksi annettiin fileUploadController ja sille määritettiin riippuvuuksiksi scope ja Upload-objektit. Upload-objekti tulee ng-file-upload-moduulin kanssa, jonka toimintaa käsitellään myöhemmin myös tässä luvussa.

index.html | uploadFilePage.js

```
1  angular
2      .module('app', ['ngFileUpload', 'ngclipboard', '720kb.datepicker'])
3      .controller('fileUploadController', fileUploadController);
4
5      fileUploadController.$inject = ['$scope', 'Upload'];
6
7      function fileUploadController ($scope, Upload) {
8
```

Kuva 22. Tiedoston lähetys -sivun ohjaimen määritykset

4.1.1 Tiedoston valitseminen

Tiedoston valitsemiseen haluttiin saada normaalin hakemistosta etsimisen lisäksi drag and drop -ominaisuus, joka tarkoittaa, että käyttäjä voi valita haluamansa tiedoston myös raahaamalla sen hakemistosta sovelluksen tiputuslaatikkoon. Tämän toiminnon toteuttamiseksi otettiin käyttöön Danial Faridin luoma ng-file-upload-direktiivimoduuli, joka on kirjoitettu tiedoston lähetyksiä varten. Moduuli tuo AngularJS:ään uusia direktiivejä ja toimintoja tiedoston lähetykseen sekä useita lähetettävän tiedoston validointiin liittyviä direktiivejä. Suurin osa moduulin direktiiveistä merkitään ngf alkuisesti. Moduuli voidaan ladata NPM:n avulla, jonka jälkeen se täytyy asettaa AngularJS-moduulin riippuvuudeksi sekä linkittää sivulle script-tageilla. (Farid 2016.)

Index.html-tiedosto on sovelluksen etusivu, jolta tiedosto lähetetään. Tiedoston valitsemista varten luotiin elementti, jonka avulla haluttu tiedosto voidaan valita. Kuvassa 23 on luotuna div-elementti, jolle on annettu dropbox-niminen luokka. Tämä luokka kertoo sovellukselle, että kyseessä on tiputuslaatikko. Luokka mahdollistaa myös tiputuslaatikkokohtaisten direktiivien käytön.

Ngf-drop ja select-direktiivit määrittävät, että tiedoston voi valita joko hakemalla tai raahaamalla. Seuraavat -disabled loppuiset direktiivit poistavat edellä mainitut toiminnot käytöstä, mikäli isDisabled-muuttuja saa arvon true. Nämä toiminnot tulevat käyttöön, kun tiedosto lähetetään palvelimelle, jotta käyttäjä ei voi valita tiedostoa enää sen jälkeen. Seuraava direktiivi on AngularJS:n oma ng-model. Direktiivi lisää malliin upFile-nimisen muuttujan, joka saa arvoksi tiedoston, jota ollaan lähettämässä. Direktiivi ngf-

multiple määrittää voidaanko valita useampi tiedosto lähetettäväksi vai vain yksi. Tässä tapauksessa tarve oli, että vain yksi tiedosto voidaan lähettää kerralla.

Accept ja ngf-pattern-direktiivit määrittävät, minkä tyyppiset tiedostot ovat sallittuja. Tähti merkintä tarkoittaa, että kaikki tiedostot ovat sallittuja. Tarve oli myös rajoittaa, kuinka isoja tiedostoja voitaisiin lähettää palvelimelle. Sovelluksen alkuperäisessä versiossa rajaksi oltiin asetettu kaksi gigatavua, joten samalla rajoituksella jatkettiin myös uudessa versiossa. Tiedostokoko on rajoitettu kahteen gigatavuun ngf-max-size-direktiivillä megatavujen tarkkuudella.

Viimeinen direktiivi ng-change kuuntelee muutoksia div-elementissä. Käytännössä se toimii siten, että kun upFile-muuttujan arvo muuttuu, se kutsuu ohjaimeen kirjoitettua funktiota nimeltään showFileName, jolle annetaan parametrina upFile-muuttuja. Funktio muuttaa dropBoxTxt-nimisen muuttujan arvoksi merkkijonon, joka sisältää valitun tiedoston nimen ja koon. Tämän jälkeen edellä mainittu muuttuja tulostetaan tiputuslaatikkoon lausekkeen avulla.

```
53          <div id="dropBox"
54          class="drop-box"
55          ng-value="dropBoxTxt"
56          ngf-drop
57          ngf-select
58          ngf-select-disabled="isDisabled"
59          ngf-drop-disabled="isDisabled"
60          ng-model="upFile"
61          name="file"
62          ngf-drag-over-class="'dragover'"
63          ngf-multiple="false"
64          ngf-allow-dir="true"
65          accept="*"
66          ngf-pattern="'*'"
67          ngf-max-size="2000MB"
68          ng-change="showFileName(upFile)">
69          {{dropBoxTxt}}</div>
```

Kuva 23. Tiputuslaatikko-elementti ja määritykset

4.1.2 Latauskertojen ja vanhentumispäivämäärän määritys

Tiedoston mahdollisten latauskertojen määritys säilytettiin samanlaisena, kuin alkuperäisessä sovelluksessa. Käyttäjä voi edelleen määrittää latauskerrat yhden ja sadan väliltä. Latauskertojen määritykseen käytettiin tavallista, numerotyyppistä input-elementtiä, jonka minimi ja maksimi arvoiksi annettiin yksi ja sata (Kuva 24).

```
94      <input id="dlTimesSetter" name="dlsetter" placeholder="1-100"
       ng-model="setdlTimes" type="number" min=1 max=100 required/>
```

Kuva 24. Latauskertojen määrityksen elementti

Sovellukseen haluttiin lisätä mahdollisuus määrittää koska käyttäjän lähettämä tiedosto vanhenee, jolloin se poistetaan palvelimelta eikä se ole enää ladattavissa. Tämän avulla tallennustila ei täyty unohdetuista tiedostoista ja se pysyy siistinä. Päivämäärän valitsemiseen haluttiin käyttää kalenterivalitsinta. Ensin yritettiin käyttää HTML5:n omaa kalenterivalitsinta, mutta se ei toiminut kaikilla selaimilla yhdenmukaisesti. Ongelmakohta oli, että joillakin selaimilla kalenterivalitsin oli käyttökelvoton, eikä siihen voitu implementoida tarvittuja ominaisuuksia. Lopulta päädyttiin käyttämään 720kb:n AngularJS-direktiivimoduulia. Direktiivi generoi oman kalenterivalitsimen input-elementtiin, jolle voidaan antaa useita eri kalenterikohtaisia määrityksiä. (720kb 2014.)

Kalenteridirektiivi otettiin, käyttöön ensin asettamalla se sovelluksen riippuvuudeksi sekä linkittämällä sen tiedosto sivulle script-tageilla. Kalenterivalitsin luodaan lisäämällä input-tekstielementti datepicker-elementin sisään. Vaatimuksena vanhenemispäivämäärälle oli, että käyttäjä voisi valita päivämäärän seuraavasta päivästä 30 päivän päähän. Rajojen asettamiseksi käytettiin date-min-limit ja date-max-limit määrityksiä (Kuva 25). Määritysten arvot lasketaan ohjaimessa (Kuva 26), jonka jälkeen ne interpoloidaan lausekkeiden avulla. Määritykset rajaavat kalenteria siten, että rajojen ulkopuolelta ei voi valita päivämäärää. (720kb 2014.)

```
101  <datepicker id="datePicker" date-format="EEE dd MMM yyyy"
102  date-min-limit="{{dateMinLimit}}" date-max-limit="{{dateMaxLimit}}">
103      <input id="datePickerInput" name="datepicker" ng-readonly="true"
104      ng-model="expDate" type="text" placeholder="dd mm yyyy" maxlength="20" required/>
105      <span id="datePickerTxt">Expiration Date
106        <span class="error" ng-show="uploaderForm.datepicker.$error.required">*</span>
107      </span>
108      <br>
109      <span>Set the expiration date for the file (max. 1 month)</span>
110  </datepicker>
```

Kuva 25. Vanhenemispäivämäärän kalenterivalitsimen elementti

```
24              var dateNow = new Date();
25              // MIN
26              var minDate = dateNow.toISOString().slice(0,10);
27              $scope.dateMinLimit = minDate;
28              // MAX
29              dateNow.setDate(dateNow.getDate() + 30);
30              var maxDate = dateNow.toISOString().slice(0,10);
31              $scope.dateMaxLimit = maxDate;
```

Kuva 26. Päivämäärä rajojen laskenta

4.1.3 Lähettäjän sähköpostin syöttö

Alkuperäisessä sovelluksessa ei ollut minkäänlaista toimintoa todentaa, että kuka tiedostoa on lähettämässä. Uuteen versioon lisättiin tekstikenttä, johon käyttäjän on pakko syöttää sähköpostiosoitteensa, ennen kuin hän voi lähettää tiedoston. Tämä ominaisuus haluttiin lisätä, että tietokantaan saadaan jokin jälki siitä, kuka kyseisen tiedoston on lähettänyt. Ominaisuuden rinnalle lisättiin myös mahdollisuus valita, haluaako käyttäjä sähköposti-ilmoituksen, kun hänen lähettämänsä tiedosto on ladattu.

Kuvassa 27 on käyttäjän sähköpostia ja sähköposti-ilmoitusta varten luodut elementit. Sähköpostin syöttämisen kannalta oli tärkeää käyttää ng-pattern-direktiiviä. Direktiivillä voidaan määrittää, minkälainen merkkijono voidaan syöttää tekstikenttään. Vaatimukseksi määritettiin tavallinen sähköpostiosoitteen formaatti, jotta käyttäjä ei voi syöttää tekstikenttään ihan mitä tahansa. Sähköposti-ilmoitusta varten käytettiin checkbox-tyyppistä input-elementtiä.

```
119   <input id="uploaderEmailInput" type="text"
120   name="uploaderEmail" ng-model="uploaderEmailAddress"
121   ng-pattern="/^((([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6}))?)*$/"
122   placeholder="example@ambientia.fi" maxlength="254" required />
123   <br>
124   <label><input type="checkbox" ng-model="notifyByEmail">
125   Send me a notification when the file has been downloaded
126   </input></label>
```

Kuva 27. Lähettäjän sähköpostin syöttämisen elementti

4.1.4 Latauslinkin lähetys sähköpostilla

Uutena ominaisuutena sovellukseen lisättiin mahdollisuus lähettää palvelimelle lähetetyn tiedoston latauslinkki sähköpostilla. Käyttäjä voi lähettää linkin useaan eri osoitteeseen ja antamaan viestille aiheen sekä viestin. Sähköpostin lähetyksen kannalta ainoaksi pakolliseksi kentäksi määritettiin vastaanottajan sähköpostiosoite. Sovellus generoi oletusaiheen ja -viestin, jos kentät jätetään tyhjiksi. Sähköpostilomaketta varten tehtiin painike, joka avaa tai sulkee sen. Tämä määrittää haluaako käyttäjä lähettää sähköpostia. Lomakkeen ollessa auki käyttäjä ei voi lähettää tiedostoa ennen kuin hän on syöttänyt ainakin yhden sallitussa formaatissa olevan sähköpostiosoitteen. Jos lomake on kiinni, sovellus tulkitsee sen siten, että käyttäjä ei halua lähettää sähköpostia ja lomakkeen kentät tyhjätään. Ohjaimeen tehtiin funktio, joka seuraa lomakkeen aukioloa.

Kuvassa 28 on sähköpostilomakkeen elementit. Vastaanottajien sähköpostiosoitteiden tekstikentässä käytettiin ng-pattern-direktiiviä samalla tavalla, kuin lähettäjän osoitteen kanssa. Ainoana erona on se, että käyttäjä voi syöttää useita sähköpostiosoitteita, kunhan ne on eroteltu puolipisteillä.

```
<a href="#" ng-click="toggleEmail()" ng-value="toggleEmailTxt">{{toggleEmailTxt}}</a>
<br>
<form id="emailForm" name="emailForm" ng-hide="!showEmail">
    <input id="emailReceiversInput" class="email"
    type="text" name="receiversInput" ng-model="emailReceivers"
    ng-pattern="/^((([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6}))\;?)*$/"
    placeholder="To: (separate multiple emails with a semicolon)" required />
    <span class="error"
    ng-show="emailForm.receiversInput.$error.required
    || emailForm.receiversInput.$error.pattern">*</span>
    <br>
    <input ng-model="emailSubject" type="text" placeholder="Subject:" maxlength="78"/>
    <br>
    <textarea ng-model="emailMessage" placeholder="Message:" maxlength="800"></textarea>
</form>
```

Kuva 28. Sähköpostilomakkeen elementit

4.1.5 Tiedoston lähettäminen

Tiedoston lähettämisen kannalta pakollisiksi syötteiksi määritettiin tiedoston valitseminen, latauskertojen määritys, vanhentumispäivämäärän valinta sekä lähettäjän sähköpostin syöttäminen. Latauslinkin lähettäminen sähköpostilla on vaihtoehtoista. Vastaanottajien sähköpostiosoitteiden syöttäminen on pakollista, mikäli linkkiä ollaan lähettämässä sähköpostilla. Sovellus ohjelmoitiin siten, että käyttäjä ei voi lähettää tiedostoa ennen kuin edellä mainitut syötteet ovat annettu vaaditulla tavalla. Muussa tapauksessa tiedoston lähetys -painike on poissa käytöstä. Painikkeeseen lisättiin ng-disabled-direktiivi, johon voidaan asettaa ehtoja, jolloin painike poistuu käytöstä. Direktiivi seuraa käyttäjän syötteitä ja muuttaa painikkeen tilaa sen mukaan. Kuvassa 29 direktiivin ehdoiksi on pääasiassa määritetty, että pakolliset syötteet eivät saa olla tyhjiä, väärässä muodossa tai liian isoja. Painikkeelle on annettu myös ng-click-direktiivi, joka kutsuu uploadFile-funktiota, kun painiketta painetaan sen ollessa mahdollista. Tämä funktio saa parametrinaan tiedoston, joka ollaan lähettämässä.

```
<button id="sendButton"
ng-disabled="setdlTimes > 100 || setdlTimes < 1 ||
setdlTimes ==  null || upFile == null || isDisabled ||
(emailReceivers == null && showEmail) ||
expDate == null || uploaderEmailAddress == null"
ng-click='uploadFile(upFile)' ng-hide="fileSent">
SEND</button>
```

Kuva 29. Tiedoston lähetys -painike

Tiedoston lähetykseen kirjoitettu funktio hyödyntää ng-file-upload-moduulin upload-funktiota. Funktiossa määritetään upload-objektissa URL-osoite mihin objekti lähetetään sekä metodi, millä se lähetetään. Objektin sisällä annetaan myös data-objekti, joka sisältää lähetettävän tiedoston sekä muita arvoja, joita tahdotaan lähettää tiedoston mukana. (Farid 2016.)

Kuvassa 30 on lähetettävän objektin määritykset. Tiedosto lähetetään /upload-osoitteeseen POST-metodilla. Palvelimelle kirjoitettiin koodi, joka käsittelee /upload-osoitteeseen saapuvan liikenteen. Tiedoston lähetyksen mukana kulkeutuvat sille asetetut latauskerrat, vanhentumispäivämäärä, lähettäjän sähköpostiosoite, boolean-arvo haluaako käyttäjä ilmoituksen latauksesta sekä mahdolliset sähköpostilomakkeen syötteet. Kaikki käyttäjän syötteet lisättiin scope-objektiin ng-model-direktiivin avulla, joten ne ovat saatavilla mallista myös ohjaimen puolella.

```
60  $scope.uploadFile = function(file) {
61      // Upload file and other data
62      file.upload = Upload.upload({
63          url: '/upload',
64          method: "POST",
65          data: {
66            file: file,
67            setDLtimes: $scope.setdlTimes,
68            expDate: $scope.expDate,
69            uploaderEmail: $scope.uploaderEmailAddress,
70            notifyByEmail: $scope.notifyByEmail,
71            emailReceivers: $scope.emailReceivers,
72            emailSubject: $scope.emailSubject,
73            emailMessage: $scope.emailMessage
74          }
75      });
```

Kuva 30. Tiedoston lähetys -funktio

Kun tiedosto on lähetetty, Node.js-palvelinsovellus lähettää vastauksena generoidun linkin, mikäli tiedoston vastaanottaminen onnistui. Linkki tulostetaan tiedoston tiputuslaatikkoon ilman erillistä sivun latausta. Vikatilanteessa, mikäli tiedoston lähetys epäonnistuu, palvelin lähettää virheviestin ja tulostaa sen samaan paikkaan. Kuvassa 31 on funktio, kun lähetys on onnistunut. Funktio sisältää pääasiassa sovelluksen ulkoasun muutoksiin liittyviä asioita. Tärkeimpänä se muuttaa dropBoxTxt-muuttujan arvoksi palvelimelta vastaanotetun datan, joka sisältää generoidun latauslinkin. Tämä muuttuja tulostetaan tiputuslaatikkoon.

```
77    file.upload.then(function (response) {
78
79        dropbox.classList.add("dropbox-done");
80        progcircle.classList.add("circle-upload-done");
81        innercircle.classList.add("circle-upload-done");
82        $scope.dropBoxTxt = response.data.message;
83        $scope.dropBoxHeaderTxt = "Upload completeted!";
84        $scope.uploadDone = true;
85        $scope.progressTxt = "\u2713";
```

Kuva 31. Lähetyksen onnistumisen käsittelevä funktio

Kuvassa 32 on funktio, joka hoitaa virhetilanteet, mikäli lähetys epäonnistuu. Virhetilanteen sattuessa toiminta on täysin samanlainen, kuin onnistuessa. Ainoa ero on ulkoasun muutokset ja palvelimelta vastaanotettu data tulee sisältämään virheviestin. Virhetilanteet hoitava funktio on lähetyksen kannalta pakollinen.

```
88  }, function errorCallback (response) {
89      if (response.status > 299 || response.status < 200) {
90          dropbox.classList.add("dropbox-done");
91          progcircle.classList.add("circle-upload-failed");
92          innercircle.classList.add("circle-upload-failed");
93          $scope.dropBoxTxt = response.data.message;
94          $scope.dropBoxHeaderTxt = "Upload failed...";
95          $scope.uploadDone = true;
96          $scope.progressTxt = "\u274C";
97      }
```

Kuva 32. Lähetyksen epäonnistumisen käsittelevä funktio

4.2 Tiedoston vastaanotto ja tallennus

Node.js:n puolelle kirjoitettiin reittimoduuli uproutes.js-tiedostoon. Moduuliin on määritetty reitti, joka käsittelee käyttäjän lähettämän tiedoston vastaanoton ja tallennuksen. Samassa yhteydessä sovellus hoitaa vastaanotetun datan validoinnin, datan tallennuksen tietokantaan, mahdollisen sähköpostin lähetyksen sekä lopuksi generoidun latauslinkin välityksen käyttäjälle.

4.2.1 Tallennus Amazon S3:en

Tiedosto lähetetään /upload-osoitteeseen POST-metodilla, joten palvelimelle kirjoitettiin reittimoduuli, joka käsittelee tiedoston ja vastaa pyyntöön. Tiedoston tallennuspaikaksi valittiin Amazon S3, koska tiedostoja ei voida tallentaa paikallisesti projektin juureen vähäisen tallennustilan sekä sovelluksen skaalautumisen takia. Tiedostoilla on oltava vain yksi jaettu tal-

lennustila. Amazon S3 on rajapinta, jonka kautta voi tallentaa ja hakea tiedostoja. Amazon kutsuu tallennettuja tiedostoja nimellä key ja niiden tallennustiloja nimellä bucket. (Amazon Web Services 2016a.)

Tallennus S3:en hoidettiin multer, multer-s3, aws-sdk-moduulien avulla. Multer on moduuli multipart/form-datan käsittelyyn, jota pääasiassa käytetään tiedostojen tallennuksessa (Multer 2016). Multer-s3 laajentaa multerin kirjastoa, jotta S3 kohtaisten parametrien käyttö on mahdollista. Aws-sdk tarjoaa valmiin JavaScript-kirjaston S3 rajapinnan kanssa kommunikoimiseen (Amazon Web Services 2016b).

Tallennettavaa tiedostoa varten kirjoitettiin oma moduuli erilliseen multer.js-tiedostoon. Moduuli määrittää S3:en tallennukseen tarvittavat tunnukset ja tallennettavan tiedoston asetukset. Kuvassa 33 annetaan ensin S3 tallennustilan tunnukset. Tämän jälkeen määritetään storage-muuttujaan tallennustilan ja tallennettavan tiedoston asetukset. Asetuksissa määritetään, minkä nimiseen tallennustilaan eli bucketiin tiedosto tallennetaan sekä millä nimellä eli mikä on tiedoston key. Jokainen tiedosto tallennettaan omaan kansioonsa, joka saa uniikin hash-koodin. Tämän avulla säästytään päällekkäisyyksiltä ja tiedostot saavat uniikin tunnisteen. Lopuksi upload-muuttujaan liitetään objekti, joka sisältää tallennustilan asetukset, tiedostoa koskevat rajoitukset sekä määrityksen, kuinka monta tiedostoa tallennetaan. Kirjoitettua moduulia käytetään asynkronisen funktion (Kuva 34) avulla, johon on liitetty virheentarkistus. Saman asynkronisen funktion sisällä hoidetaan tietokanta kyselyt sekä muut toiminnot, mikäli tiedoston tallennus ei palauta virhettä.

multer.js uproutes.js

```
var multer = require('multer'),
    multerS3 = require('multer-s3'),
    aws = require('aws-sdk'),
    crypto = require('crypto'),
    env = process.env,
    s3 = new aws.S3({ accessKeyId: env.AWS_ACCESS_KEY_ID,
                      secretAccessKey: env.AWS_SECRET_ACCESS_KEY });

var maxFileSize = (1000 * 1000 * 1000) * 2; // 2GB

// Configs for the uploaded files
var storage = multerS3({
    s3: s3,
    bucket: env.S3_BUCKET,
    key: function (req, file, cb) {
        var folderHash = crypto.randomBytes(32).toString('hex');
        var filename = file.originalname;
        cb(null, folderHash + "/" + filename);
    }
});

// "packaged" multer settings
var upload = multer({ storage : storage , limits: { fileSize: maxFileSize }}).single('file');

exports.upload = upload;
```

Kuva 33. Tiedoston tallennus -moduuli

```
12  router.route('/upload')
13
14    .post(function (request, response) {
15
16        multer.upload(request, response, function(err) {
17            if (err) {
18                logger.error("ERROR ! File upload failed.", err);
19                return response.status(400).send({message : "ERROR ! File upload failed."});
20            }
```

Kuva 34. Tallennus funktion kutsuminen

4.2.2 Tietojen tallennus tietokantaan

Sovelluksen toiminnan kannalta oli olennaista, että jokaisesta tiedostosta jää merkintä tietokantaan. Tietokantaan lisätään tiedoston tunnistetiedot, joiden avulla se voidaan ladata ja poistaa sekä päivittää sen tietoja. Node.js:lle on olemassa PostgreSQL-tietokantojen käyttöä varten oleva, node-postgres-moduuli. Moduuli tarjoaa ominaisuudet ja toiminnot PostgreSQL-tietokantojen käsittelyyn Node.js-alustalla (Carlsson 2016). Tietokantaa varten kirjoitettiin oma palvelumoduuli db.js-tiedostoon, joka ottaa yhteyden tietokantaan. Kuvassa 35 otetaan käyttöön pg-moduuli, jonka avulla luodaan client-objekti. Objektille annetaan ympäristömuuttujina tietokantaan yhdistämiseen tarvittavat tiedot, jonka jälkeen connect-funktiolla yritetään yhdistää tietokantaan. Yhteyden muodostettua, moduuli tarjoaa client-objektin, jonka avulla voidaan tehdä kyselyjä ja komentoja tietokantaan muissa sovelluksen osissa (Kuva 36).

```
1  var pg         = require('pg'),
2      logger     = require('../../utils/logger'),
3      env        = process.env,
4      conString  = env.OPENSHIFT_POSTGRESQL_DB_URL + env.OPENSHIFT_APP_NAME,
5      client     = new pg.Client(conString);
6
7  // Connect to the database
8  client.connect(function(err) {
9      if(err) {
10       return logger.error('Could not connect to database', err);
11     }
12     logger.info("Connected to database.");
```

Kuva 35. Tietokantaan yhdistäminen

```
25  // Export the database client for the routes to use
26  exports.client = client;
```

Kuva 36. Client-objekti asetettuna jakoon

Tiedostojen vastaanottamisen ja tallentamisen lisäksi, multer lisää request-objektiin body, file ja files-objektit, joiden avulla voidaan ottaa tietoja talteen tallennettavasta tiedostosta (Multer 2016). Tiedostoon liittyvät tiedot, kuten sille määritetty nimi ja tiedoston polku, löytyvät file-objektin alta.

Käyttäjän antamat syötteet ovat body-objektin alla. Kaikki tiedostoon liittyvät tiedot asetettiin samaan uploadData-objektiin (Kuva 37), jotta koodi säilyisi siistinä ja tietojen käyttäminen olisi helpompaa.

```
22  var uploadData = {fileHash: crypto.randomBytes(32).toString('hex'),
23                    fileName: request.file.originalname,
24                    filePath: request.file.key,
25                    fileSetDLtimes: request.body.setDLtimes,
26                    fileExpDateMillis: Date.parse(request.body.expDate),
27                    fileUploaderEmail: request.body.uploaderEmail,
28                    notifyByEmail: request.body.notifyByEmail,
29                    emailReceivers: request.body.emailReceivers,
30                    emailSubject: request.body.emailSubject,
31                    emailMessage: escape(request.body.emailMessage)};
```

Kuva 37. Tiedoston data-objekti

Ennen tiedoston tietojen tallennusta tietokantaan, ne ovat hyvin tärkeää validoida haitallisten arvojen varalta. Haitalliset syötteet voivat aiheuttaa sovelluksen kaatumisen. Jos validointi ei mene läpi, tiedoston tallennus perutaan ja sovellus lähettää käyttäjälle virheviestin. Tietojen validointiin löytyi muutamia valmiita kirjastoja, mutta opin kannalta päädyttiin toteuttamaan oma validointi moduuli. Moduulissa tarkistetaan, että käyttäjän syötteet ovat samoissa sallituissa rajoissa kuin käyttäjäpuolellakin on määritetty.

Kun tiedot on validoitu hyväksytysti, ne lisätään tietokantaan. Tietokanta kyselyitä ja komentoja varten luotiin oma moduuli nimeltään dbqueries.js, johon kirjoitettiin jokainen sovelluksen tarvitsema kysely ja komento. Moduulista kutsuttiin insertFileQuery-funktiota, jolle annettiin parametrina tiedoston tiedot sisältävä objekti. Funktio sisältää tavallisen SQL Insert -komennon, joka lisää uuden rivin tietokantaan tiedoston tiedoilla. (Kuva 38.)

```
13  db.client.query(SQL`INSERT INTO files
14  (fileName, hash, path, setDLtimes, dlTimesLeft, expDateMillis, uploaderEmail, notifyByEmail)
15  VALUES (${uploadData.fileName}, ${uploadData.fileHash}, ${uploadData.filePath},
16          ${uploadData.fileSetDLtimes}, ${uploadData.fileSetDLtimes},
17          ${uploadData.fileExpDateMillis}, ${uploadData.fileUploaderEmail},
18          ${uploadData.notifyByEmail})`,
```

Kuva 38. Insert-komento tietojen tallennusta varten

4.2.3 Sähköpostin lähetys

Tiedoston latauslinkin lähettäminen sähköpostilla tehtiin vaihtoehtoiseksi toiminnoksi. Jotta toimintoa voidaan käyttää, käyttäjän täytyy syöttää vähintään yksi validi sähköpostiosoite tai muussa tapauksessa sähköpostin vastaanottajille tarkoitettu muuttuja saa tyhjän arvon. Kun tallennetun tiedoston tiedot on lisätty tietokantaan, tarkistetaan, onko käyttäjä antanut vastaanottajille sähköpostiosoitteita eli onko muuttujan arvo jokin muu kuin tyhjä. Jos muuttuja ei ole tyhjä, lähetetään sähköposti.

Latauslinkin sähköpostia varten luotiin oma moduuli, johon kirjoitettiin sähköpostin lähetys -funktio. Funktiolle syötetään parametrina vastaanotetun tiedoston tiedot sisältävä objekti sekä generoitu latauslinkki. Funktion rakentamiseen käytettiin Nodemailer-moduulia, joka on tarkoitettu sähköpostin lähettämiseen. Lähetetty sähköposti sisältää tiedot ladattavasta tiedostosta sekä tiedoston latauslinkin. Aihetta ja viestiä varten tehtiin oletusmääritykset, jos käyttäjä on päättänyt jättää ne antamatta.

Kuvassa 39 on sähköpostin lähetyksen kannalta oleellisimmat asiat Nodemaileria käytettäessä. Nodemailer käyttää sähköpostin lähetyksessä transporter-objektia, jonka avulla sähköposti voidaan lähettää. Objekti saa parametrina smtpConfig-objektin, joka sisältää SMTP-sähköpostipalvelimen asetukset. Asetuksissa voidaan määrittää sähköpostipalvelimen osoite, käytettävän portin numero ja sähköpostipalvelimen tunnukset. Lopuksi sähköposti lähetetään transporter-objektin sendMail-funktiolla, jolle annetaan parametrina mailOptions-objekti. Objektissa määritetään lähetettävän sähköpostin tiedot eli lähettäjä, vastaanottaja, aihe ja viesti. (Reinman 2016.)

```
49      var transporter = nodemailer.createTransport(smtpConfig);
50
51      // E-mail setup
52      var mailOptions = {
53          from: '<' + uploadData.fileUploaderEmail + '>',
54          to: uploadData.emailReceivers,
55          subject: uploadData.emailSubject,
56          text: uploadData.emailMessage,
57          html: htmlEmailMessage
58      };
59
60      // send mail with defined transport object
61      transporter.sendMail(mailOptions, function (error, info) {
```

Kuva 39. Nodemailer-määritykset sähköpostia varten

4.3 Tiedoston lataus, päivitys ja poisto

Alkuperäisessä sovelluksessa tiedoston lataaminen onnistuu vasta erillisellä sivulla, mutta uuteen versioon lisättiin mahdollisuus ladata tiedosto myös suoraan, jos linkki lähetetään sähköpostilla. Aluksi suunnitelmana oli, että erillinen tiedoston lataussivu poistettaisiin kokonaan uudesta versiosta, mutta se päädyttiin silti säilyttämään. Syynä tälle oli lataussivun tuoma turvallisuuden ja varmuuden tunne lataajalle. Pelkkä yksittäinen linkki, joka aloittaa välittömästi latauksen sitä painettaessa ilman minkäänlaista saateviestiä, saattaa aiheuttaa epävarmuutta lataajassa. Tästä syystä tiedoston lataaminen toteutettiin siten, että sovelluksen tarjoama generoitu linkki ohjaa lataussivulle ja sähköpostin linkki aloittaa latauksen suoraan. Tällä tavoin käyttäjälle selviää aina latauslinkin alkuperä.

Tiedoston lataamista varten kirjoitettiin oma reittimoduuli downroutes.js-tiedostoon. Tiedoston lataus prosessia varten kirjoitettiin muutama eri

polku, joiden avulla hoidetaan lataussivun lähetys sekä tiedoston lataus, päivitys ja poisto. Jokainen tiedoston lataus vähentää sen latauskertoja yhdellä ja kun ne loppuvat, tiedosto poistetaan S3:sta ja tietokannasta.

4.3.1 Lataussivu

Jokaiselle tallennetulle tiedostolle annetaan uniikki hash-koodi, joka tallennetaan tietokantaan. Latauslinkit ovat muodossa ’www.esim.fi/download/p=hash-koodi’ tai ’www.esim.fi/download/f=hash-koodi’. Erona on yhtä kuin merkkiä edeltävä kirjain. P-alkuinen ohjaa lataussivulle ja F-alkuinen lataa tiedoston. Käytännössä tiedosto voidaan ladata lataussivun linkillä vain muuttamalla kirjain, mutta tämä ei ole ongelma, koska tästä voidaan olettaa, että lataajalla on jo entuudestaan käsitys linkin alkuperästä.

Tiedoston latausta varten kirjoitettiin kolme eri polkua omaan reittimoduuliin. Lataussivua varten jouduttiin kirjoittamaan kaksi eri polkua, koska response-objektilla ei pysty lähettämään tiedostoa ja data-objektia samaan aikaan. Lataussivun lähetys toteutettiin siten, että kun käyttäjä painaa linkkiä, Node.js tarkistaa linkin hash-koodin avulla tietokannasta, onko tiedostoa olemassa. Jos tiedostoa ei ole olemassa se lähettää sivun, joka ilmoittaa käyttäjälle, että tiedosto ei ole enää saatavilla. Mikäli tiedosto on vielä tietokannassa, käyttäjälle lähetetään lataussivu. Tässä vaiheessa käyttäjälle on kuitenkin lähetetty vasta sivu ilman oikean tiedoston tietoja, joilla se voidaan ladata. Lataussivua varten kirjoitettiin AngularJS-ohjain, joka lähettää heti sen latautuessa GET-pyynnön, johon palvelin vastaa lähettämällä käyttäjälle tiedoston data-objektin.

Kuvassa 40 on lataussivun reitti. Kun sovellus vastaanottaa pyynnön, se ottaa URL-osoitteen lopusta tiedoston hash-koodin request-objektista. Tietokanta kysely moduulista kutsutaan getFileDataQuery-funktiota, joka tekee kyselyn tietokantaan käyttämällä hash-koodia. Kysely on tarkoitettu tiedoston tietojen hakemiseen, mutta tässä tapauksessa, sillä tarkistetaan, onko tiedostoa olemassa.

```
11  router.route('/download/p=:hash')
12    .get(function(request, response) {
13
14        var fileHash = request.params.hash;
15
16        // This function is called here to check if the file still exists
17        dbQueryService.getFileDataQuery(fileHash, function(fileData, err) {
18            if (err) {
19                return response.sendFile(path.resolve('./public/pages/deadlink.html'));
20            }
21            response.sendFile(path.resolve('./public/pages/downloadpage.html'));
22        });
23    });
```

Kuva 40. Lataussivun polku

getFileDataQuery-funktio tekee kyselyn tietokantaan, jonka jälkeen tarkistetaan palauttaako se yhtäkään riviä sekä onko löydetyllä tiedostolla enää

latauksia jäljellä (Kuva 41). Tiedoston latauskertoja tarkistetaan tässä yhteydessä, koska on mahdollista, että tiedostoa ei ole ehditty vielä poistaa, vaikka sen latauskerrat ovat jo loppu. Syynä tälle on, että tiedoston latauskerrat päivitetään heti tiedoston latauksen alkaessa, mutta tiedosto poistetaan vasta kun lataus on valmis. Tällöin on tiedoston latauksen pituinen ikkuna, jolloin tiedoston latauskerrat ovat loppu, mutta sitä ei ole vielä poistettu. Tiedostoa ei voida myöskään päivittää vasta, kun tiedosto on ladattu, koska silloin tuon saman ikkunan aikana tiedoston lataaminen voitaisiin aloittaa uudestaan, vaikka sen latauskerrat olisivatkin jo loppu. Tässä tapauksessa, jos tarkistuksen ehdot täyttyvät, funktio palauttaa tiedoston data-objektin. Tämä on merkki siitä, että tiedosto on olemassa ja ladattavissa. Data-objektin palautus suorittaa callback-funktion, joka lähettää käyttäjälle lataussivun response-objektilla (Kuva 40). Muussa tapauksessa käyttäjälle ilmoitetaan, että tiedosto ei ole enää saatavilla.

```
db.client.query(SQL`SELECT * FROM files WHERE hash = ${fileHash} FOR UPDATE`, function(err, result) {

    if (result.rows.length > 0 && result.rows[0].dltimesleft > 0) {
        logger.info("Filedata fetched with hash " + fileHash);
        var filedata = result.rows[0];
        db.client.query('COMMIT');
        return cb(filedata, null);

    } else {
        db.client.query('ROLLBACK');
        return cb(null, true);
    }
```

Kuva 41. Tietojen haku -funktio

Lataussivua varten kirjoitettiin fileDownloadController-ohjain (Kuva 42), joka lähettää sivun latauduttua pyynnön palvelimelle. HTTP-pyyntöjä voidaan lähettää AngularJS:n sisäänrakennetun HTTP-palvelun avulla. Ohjain lähettää GET-pyynnön palvelimelle ’/download/getFileData/hash-koodi’-osoitteeseen (Kuva 43). Hash-koodi otetaan selaimen URL-osoitteen lopusta, joka on tässä vaiheessa ’/download/p=hash -koodi’.

```
angular
    .module('app', [])
    .controller('fileDownloadController', fileDownloadController);

    fileDownloadController.$inject = ['$scope', '$http'];

    function fileDownloadController ($scope, $http) {
```

Kuva 42. Lataussivun ohjain

```
        $http({
                method: 'GET',
                url: '/download/getFileData/' + hash
```

Kuva 43. Palvelimelle lähetettävä HTTP-pyyntö

Palvelin käsittelee pyynnön tekemällä pyynnön mukana tulleen hash-koodin avulla tietokantaan kyselyn, jolla haetaan tiedoston data-objekti. Tähän käytettiin samaa funktiota kuin lataussivun lähetykseen, mutta tällä kertaa palvelin vastaa käyttäjälle lähettämällä tiedoston tietoja sisältävän data-objektin. Objektiin sisältyy tiedoston hash-koodi, nimi, latauskerrat ja vanhenemispäivämäärä. (Kuva 44.) Lataussivun ohjain vastaanottaa lähetetyn data-objektin ja lisää sen mukana tulleet tiedot sivulle skoopin avulla (Kuva 45).

```
router.route('/download/getFileData/:hash')
  .get(function(request, response) {

      var fileHash = request.params.hash;

      dbQueryService.getFileDataQuery(fileHash, function(fileData, err) {
          if (err) {
              return response.sendFile(path.resolve('./public/pages/deadlink.html'));
          }

          var fileDLtimesLeft = fileData.dltimesleft;
          var fileSetDLtimes = fileData.setdltimes;
          var fileName = fileData.filename;
          var fileExpDateMillis = fileData.expdatemillis;
          var fileExpDate = new Date(parseInt(fileExpDateMillis)).toDateString();

          // Send data to view
          response.send({fileName : fileName,
                         hash : fileHash,
                         setDLtimes : fileSetDLtimes,
                         dlTimesLeft : fileDLtimesLeft,
                         expDate : fileExpDate});
      });
  });
```

Kuva 44. Tietojen lähetys lataussivulle

```
}).then (function (response, data) {
      $scope.fileName = response.data.fileName;
      $scope.dlLink = baseUrl + '/download/f=' + response.data.hash;
      $scope.setDLtimes = response.data.setDLtimes;
      $scope.dlTimesLeft = response.data.dlTimesLeft;
      $scope.expDate = response.data.expDate;
```

Kuva 45. Vastaanotettujen tietojen lisäys sivulle

4.3.2 Lataaminen ja päivitys

Tiedoston latauslinkki lähettää GET-pyynnön ’/download/f=hash -koodi’-osoitteeseen, joka aloittaa tiedoston lataamisen, mutta ennen varsinaista latausprosessin aloittamista, sovellus päivittää tiedoston latauskerrat vähentämällä niitä yhdellä. Tämän lisäksi sovellus lähettää sähköposti-ilmoituksen tiedoston latauksesta, jos tiedoston lähettänyt osapuoli on niin määrittänyt.

Kuvassa 46 on sovelluksen toiminta ennen tiedoston latausta pyynnön saapuessa. Sovellus hakee tietokannasta ne tiedot, joita tarvitaan tiedoston päivittämiseen, lataukseen ja poistoon sekä sähköpostin lähetykseen. Tietojen haun jälkeen tiedoston latauskertoja vähennetään yhdellä, jonka jälkeen vähennetty arvo päivitetään tietokantaan. Päivittämiseen käytetään tietokanta kysely -moduulin updateFileQuery-funktiota. Funktiossa käytetään UPDATE-komentoa, joka päivittää hash-koodilla haetun rivin latauskerrat (Kuva 47). Päivityksen jälkeen, kuvassa 46 tarkistetaan, onko tiedoston lähettäjä halunnut, että hänelle lähetetään sähköposti-ilmoitus tiedoston latauksesta. Sähköposti-ilmoitus toteutettiin myös Nodemailerin avulla. Ainoa ero latauslinkki-sähköpostin lähetykseen verrattuna on viestin muoto.

```
52  router.route('/download/f=:hash')
53
54    .get(function(request, response) {
55
56        var fileHash = request.params.hash;
57
58        dbQueryService.getFileDataQuery(fileHash, function(fileData, err) {
59            if (err) {
60                return response.sendFile(path.resolve('./public/pages/deadlink.html'));
61            }
62
63            var fileName = fileData.filename;
64            var fileDLtimesLeft = fileData.dltimesleft;
65            var filePath = fileData.path;
66            var fileUploaderEmail = fileData.uploaderemail;
67            var fileExpDateMillis = fileData.expdatemillis;
68            var notifyByEmail = fileData.notifybyemail;
69
70            fileDLtimesLeft--;
71
72            dbQueryService.updateFileQuery(fileDLtimesLeft, fileHash, filePath);
73
74            if (notifyByEmail == 'true') {
75                emailService.notifyByEmail(fileUploaderEmail, fileName, fileDLtimesLeft, fileExpDateMillis);
76            }
```

Kuva 46. Toiminta ennen tiedoston latausta

```
67  db.client.query(SQL`UPDATE files SET dltimesleft = ${fileDLtimesLeft} WHERE hash = ${fileHash}`,
```

Kuva 47. Latauskertojen päivitys -komento

Kun tiedoston latauskerrat on päivitetty ja mahdollinen sähköposti lähetetty, aloitetaan tiedoston lataus S3:sta. Tiedoston S3:sta latausta ja poistamista varten kirjoitettiin oma fileHandling.js-moduuli. Kuvassa 48 kutsutaan moduulin asynkronista downloadFile-funktiota, joka aloittaa latauksen S3:sta ja latauksen päätyttyä palauttaa arvon true. Callback-funktion sisällä seurataan koska lataus on valmistunut ja ovatko tiedoston latauskerrat loppuneet. Vasta kun molemmat ehdot täyttyvät, sovellus poistaa tiedoston tietokannasta ja S3:sta. Tiedoston lataamiseen ja poistamiseen käytettiin samaa AWS-SDK-moduulia kuin tiedoston tallentamiseenkin.

```
fileService.downloadFile(request, response, filePath, function(dldone) {

    // Delete file from storage and DB if out of dltimes and download ended
    if (fileDLtimesLeft <= 0 && dldone) {
        logger.audit("File ran out of downloads | File: " + filePath);
        dbQueryService.deleteFileQuery(fileHash, filePath);
        fileService.deleteFile(filePath);
    }
});
```

Kuva 48. Tiedoston lataus -funktion käyttö

Kuvassa 49 on funktio, jolla tiedosto ladataan. Tiedoston lataamiseen tarvitaan S3-tunnukset, bucketin nimi ja tiedoston polku. Tiedoston latausta varten täytyy luoda lukuvirtaus (readstream), jota pitkin tiedosto ladataan vähitellen S3:sta. Lukuvirtaus luodaan ensin hakemalla oikea tiedosto S3:sta getObject-funktiolla, joka saa parametrina tiedoston bucketin ja polun sisältävän objektin. Tämän jälkeen avataan lukuvirtaus tuohon tiedostoon. Tiedoston lataus aloitetaan pipe-funktiolla, joka saa parametrina response-objektin, jolle annetaan liitteenä oikean tiedoston polku. Tiedoston lataukselle on määritetty kaksi tapahtumaa, jolloin funktio palauttaa arvon true. Ensimmäinen on close, joka käytännössä tarkoittaa, että käyttäjä on itse lopettanut latauksen ennen sen valmistumista. Toinen on finish, joka tarkoittaa, että tiedoston lataus on valmis.

```
    AWS            = require('aws-sdk'),
    s3             = new AWS.S3({accessKeyId: env.AWS_ACCESS_KEY_ID,
                                 secretAccessKey: env.AWS_SECRET_ACCESS_KEY });

// Download file function
var downloadFile = function (request, response, filePath, cb) {

    var client_ip = JSON.stringify(request.headers["x-forwarded-for"]);

    var options = {
        Bucket    : env.S3_BUCKET,
        Key    : filePath
    };

    var fileStream = s3.getObject(options).createReadStream().on('error', function (err) {
        return logger.error("Error when trying to download. | Downloader IP: " +
        client_ip + " | File: " + filePath + " | ", err);
    });

    response.attachment(filePath);
    fileStream.pipe(response)
      .on('close', function () {
        logger.audit("Download canceled. | Downloader IP: " + client_ip + " | File: " + filePath);
        return cb(true);
    }).on('finish', function () {
        logger.audit("File downloaded. | Downloader IP: " + client_ip + " | File: " + filePath);
        return cb(true);
    });
}
```

Kuva 49. Lataus funktion rakenne

4.3.3 Poistaminen

Tiedoston poistamiseen tarvitaan myös tiedoston bucketin ja polun sisältävä options-objekti. Objekti annetaan S3:n deleteObject-funktion parametrina, jonka avulla se osaa paikantaa ja poistaa oikean tiedoston. (Kuva 50.) Tiedosto poistetaan tietokannasta DELETE -komennon avulla, jossa kohdennetaan oikean tiedoston riviin sen hash-koodilla (Kuva 51).

```
39  var deleteFile = function (filePath) {
40
41      // File location
42      var options = {
43          Bucket    : env.S3_BUCKET,
44          Key     : filePath
45      };
46
47      // Delete from S3
48      s3.deleteObject(options, function(err, data) {
49          if (err) {
50              return logger.error("Error deleting file " + filePath, err);
51          }
52          logger.audit("File deleted | File: " + filePath);
53      });
54  }
```

Kuva 50. Tiedoston poistamis -funktion rakenne

```
85          db.client.query(SQL`DELETE FROM files WHERE hash = ${fileHash}`,
```

Kuva 51. Tiedoston poistamis -komento

Vanhentuneiden tiedostojen poistoa varten tehtiin checkForExpiredFiles-funktio, jota kutsutaan setInterval-funktion avulla 12 tunnin välein. Funktiota kutsutaan sovelluksen juuressa app.js -tiedostossa, joten se toimii sovelluksen taustalla. (Kuva 52.) Jokaisen tiedoston vanhenemispäivämäärä tallennetaan tietokantaan millisekunteina. Vanhentuneita tiedostoja haetaan kyselyllä, joka hakee niiden tiedostojen rivit, joiden vanhenemispäivämäärä millisekunteina on pienempi kuin millisekunti määrä, joka on otettu sillä hetkellä, kun kysely tehdään. Jos rivejä löytyy, ne käydään läpi for-loopin avulla ja poistetaan tietokannasta sekä S3:sta. (Kuva 53.)

```
60  // Check for old files
61  setInterval(dbQueryService.checkForExpiredFilesQuery, checkIntervalMillis);
```

Kuva 52. Vanhentuneiden tiedostojen tarkistus -funktion käyttö

```
116 db.client.query(SQL`SELECT * FROM files WHERE expdatemillis < ${dateNowMillis}`,
117   function (err, result) {
118     if (err) {
119         logger.error("Error getting expired files !", err);
120         return db.client.query('ROLLBACK');
121     }
122
123     if (result.rows.length > 0) {
124
125         logger.audit("Expired file(s) found.");
126         db.client.query('COMMIT');
127
128         for (var i = 0; i < result.rows.length; i++) {
129             var fileHash = result.rows[i].hash;
130             var filePath = result.rows[i].path;
131
132             deleteFileQuery(fileHash, filePath);
133             fileService.deleteFile(filePath);
134         }
135     } else {
136         logger.info("No expired files found.");
137         return db.client.query('ROLLBACK');
138     }
139 });
```

Kuva 53. Vanhentuneiden tiedostojen tarkistus -funktion rakenne

5 YHTEENVETO

AngularJS ja Node.js ovat erittäin hyvä ja tehokas yhdistelmä yksinkertaisia ja monimutkaisempiakin verkkosovelluksia kehittäessä. Molemmat teknologiat ovat aloittelevan kehittäjän omaksuttavissa suhteellisen helposti. Niiden kattavien dokumentointien ja aktiivisten kehitysfoorumeiden avulla pääsee helposti alkuun ja selviää vastaantulevista ongelmatilanteista. AngularJS tarjoaa erittäin kattavat resurssit verkkosovelluksen käyttäjäpuolen ohjelmointiin ja hyvin pienellä työllä saa luotua dynaamisesti ja reaaliaikaisesti toimivan käyttöliittymän. Node.js on erittäin laadukas ja tehokas palvelinpuolen alusta. Suurin etu on sen pakettienhallintajärjestelmä, joka tarjoaa lähes rajattomasti paketteja, joka tarpeeseen. OpenShift on kattava pilvipalvelu, joka tarjoaa suuren valikoiman eri teknologioita sovelluksen kehittämiseen sekä pilviympäristön, joka hoitaa sovelluksen ajamisen ja monitoroinnin. Kehittäjä voi keskittyä täysin sovelluksen kehittämiseen.

Opinnäytetyön myötä saatiin toteutettua suunnitelmien mukainen, lähes julkaisu valmis uusi versio alkuperäisestä sovelluksesta, joka miellyttää itse tekijää ja toimeksiantajaa. Alkuperäiseen sovellukseen verratessa uusi versio on huomattavasti edistyneempi käyttöliittymältään ja toiminnoiltaan. Uuden version käyttöliittymä on dynaaminen ja koko tiedoston palvelimelle lähettämisen prosessi voidaan tehdä ilman erillisiä sivun latauksia, mikä tekee sovelluksen käytöstä sujuvaa. Lähetykseen liitetyt uudet valinnat tuovat monipuolisuutta, miten tiedosto voidaan toimittaa eteenpäin ja minkälaisia määrityksiä tiedostolle voidaan antaa. OpenShift hoitaa sovelluksen käyttöönoton, ajamisen sekä sovelluksen horisontaalin skaalaamisen. Yhdellä palvelimella ajettavaa monoliittista sovellusta ei voida skaalata sen saumattomuuden vuoksi kuin tehostamalla palvelinkonetta fyysisesti, joka voi käydä kalliiksi. Verratessa uutta sovellusta vanhaan voi todeta, että AngularJS ja Node.js ovat ehdottomasti hyviä valintoja. Lopputuotteena saatiin luotua vaatimukset täyttävä sovellus, joka hyödyntää ja demonstroi sen toteutukseen valittujen tekniikoiden parhaita puolia.

Opinnäytetyö on ollut iso ja mielenkiintoinen oppimiskokemus kokonaisuudessaan. Vaikka kyseessä oli suhteellisen yksinkertainen ja pienikokoinen sovellus ei ongelmilta kuitenkaan säästytty. Työn alussa tekijällä ei ollut yhtään kokemusta työssä käytetyistä teknologioista, joten työ sisälsi paljon uuden opiskelua. Kaikki oli niin uutta, että työn alkuun pääseminen aiheutti ongelmia ja epävarmuutta ratkaisujen tekemisessä kehityksen suhteen. Ongelmista huolimatta sovellus saatiin opinnäytetyön aikana lähes julkaisu valmiiksi. Työn myötä kehittäjä tuntee itsensä kehittyneen hyvin paljon verkkosovellusten kehittäjänä, niin AngularJS, Node.js ja OpenShiftin osalta, kuin myös yleisellä tasolla.

LÄHTEET

720kb. 2014. Angular Datepicker. Viitattu 21.7.2016. https://github.com/720kb/angular-datepicker

Amazon Web Services. 2016a. Viitattu 28.7.2016 http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html

Amazon Web Services. 2016b. AWS SDK for JavaScript in Node.js. Viitattu 29.7.2016. https://aws.amazon.com/sdk-for-node-js/

Ambientia. 2016. Viitattu 8.8.2016. https://www.ambientia.fi/

Austin Andrew. 2014. An overview of AngularJS for managers. Viitattu 9.5.2016 http://andrewaustin.com/an-overview-of-angularjs-for-managers/

Bégaudeau Stéphane. 2014. Everything you need to understand to start with AngularJS. Viitattu 9.5.2016 http://stephanebegaudeau.tumblr.com/post/48776908163/everything-you-need-to-understand-to-start-with

Carlsson Brian. 2016. Node-postgres. Viitattu 1.8.2016. https://www.npmjs.com/package/pg

Farid Danial. 2016. Ng-file-upload. Viitattu 21.7.2016 https://github.com/danialfarid/ng-file-upload

Google. 2016a. Templates. Viitattu 14.5.2016 https://docs.angularjs.org/guide/databinding

Google. 2016b. Data Binding. Viitattu 14.5.2016 https://docs.angularjs.org/guide/templates

Google. 2016c. Directive. Viitattu 14.5.2016 https://docs.angularjs.org/guide/databinding

Google. 2016d. Controller. Viitattu 9.5.2016 https://docs.angularjs.org/guide/controller

Google. 2016e. Scope. Viitattu 9.5.2016 https://docs.angularjs.org/guide/scope

Google. 2016f. Expression. Viitattu 9.5.2016 https://docs.angularjs.org/guide/expression

Google. 2016g. Module. Viitatu 19.6.2016 https://docs.angularjs.org/guide/module

Google. 2016h. Dependency Injection. Viitatu 19.6.2016 https://docs.angularjs.org/guide/di

Martínez Daniel Pecos. 2013. Node.js and V8 history. Viitattu 14.7.2016. https://danielpecos.com/2013/12/18/nodejs-v8-history/

Multer, 2016. Multer. Viitattu 29.7.2016. https://github.com/expressjs/multer

Nitin Shirastava. 2014. Introduction to AngularJS. Viitattu 9.5.2016 http://www.codeproject.com/Articles/803294/Part-Introduction-to-AngularJS

OpenShift. 2016a. Basic Terminology. Viitattu 2.8.2016. https://developers.openshift.com/overview/basic-terminology.html

OpenShift. 2016b. Viitattu 19.7.2016. https://www.openshift.com/

Oracle. n.d.a. What is Java? Viitattu 2.5.2016. https://java.com/en/download/faq/whatis_java.xml

Oracle. n.d.b. How Will Java Technology Change My Life? Viitattu 2.5.2016. https://docs.oracle.com/javase/tutorial/getStarted/intro/changemylife.html

Reinman Andris. 2016. Nodemailer. Viitattu 1.8.2016 https://nodemailer.com/

Salonen Jari. 2012. Johdanto JavaScript-sovellusten kehitykseen Node.js:llä. Viitattu 14.7.2016. http://blite.iki.fi/artikkelit/javascript-nodejs-johdanto/

Singh Ankit. 2015. Top 4 Javascript Concepts a Node.js Beginner Must Know. Viitattu 14.7.2016. https://simpleprogrammer.com/2015/12/04/top-4-javascript-concepts-a-node-js-beginner-must-know/

Strongloop. 2016a. Viitattu 28.7.2016. https://expressjs.com/

Strongloop. 2016b. Basic Routing. Viitattu 28.7.2016. https://expressjs.com/en/starter/basic-routing.html

Strongloop. 2016c. Express API. Viitattu 28.7.2016. http://expressjs.com/en/api.html

Tutorialspoint. n.d.a. Java Overview. Viitattu 5.5.2016. http://www.tutorialspoint.com/java/java_overview.htm

Tutorialspoint. n.d.b. AngularJS MVC Architecture. Viitattu 14.5.2016. http://www.tutorialspoint.com/angularjs/angularjs_mvc_architecture.htm