

AngularJS-sovelluksen päivittäminen Angular-sovellukseksi

Lasse Lindén

Opinnäytetyö

Syyskuu 2019

Liiketalouden ala

Tradenomi (AMK), tietojenkäsittelyn tutkinto-ohjelma

Tekijä(t) Lindén, Lasse	Julkaisun laji Opinnäytetyö, AMK	Päivämäärä Syyskuu 2019
	Sivumäärä 49	Julkaisun kieli Suomi
		Verkojulkaisulupa myönnetty: x
Työn nimi AngularJS-sovelluksen päivittäminen Angular-sovellukseksi		
Tutkinto-ohjelma Tietojenkäsittelyn tutkinto-ohjelma		
Työn ohjaaja(t) Tommi Tuikka		
Toimeksiantaja(t) Protacon Solutions Oy		
<p>Tiivistelmä</p> <p>Nykyaikainen front-end-sovelluskehitys kehittyy nopeasti. Sovelluskehityksen apuna käytetään usein erilaisia sovelluskehyskiä, jotka tarjoavat valmiita työkaluja kehittäjille. Front-end-puolen nopean kehityksen seurauksena myös nämä sovelluskehyski vanhenevat melko nopeasti. Angular on alusta alkaen uudelleenkirjoitettu web-sovelluskehys ja se on suosittu AngularJS-sovelluskehyski seuraaja. Angular on uudelleenkirjoitettu, jotta se pystyy vastaamaan nykyaikaisten sovellusten tarpeisiin karsimalla heikkoja osia pois.</p> <p>Tutkimuksen tavoitteena oli löytää mahdollisimman helppo keino päivittää olemassa oleva AngularJS-sovellus Angular-sovellukseksi. Tutkimus toteutettiin kehittämistutkimuksena, sillä käytännön työn kautta saatiin tärkeää tutkimustietoa päivityksen eri vaiheista. Kehittämistutkimuksen avuksi hankittiin tarvittavaa teoretista tietoa sovelluskehyskiistä ja niiden eroavaisuuksista. Teoretisen tiedon hankkimiseen käytettiin pääasiassa verkkolähteitä.</p> <p>Tutkimustuloksena syntyi vaihe vaiheelta ohjeistus siihen, kuinka tutkimuskohteena ollut sovellus voidaan päivittää Angular-sovellukseksi. Ohjeistuksen lisäksi kerättiin tärkeää tietoa sovelluskehysten välisistä eroavaisuuksista. Tietoa näistä eroavaisuuksista voidaan käyttää perehdytystarkoituksessa siirryttäessä AngularJS-projektista Angular-projektiin.</p> <p>AngularJS-sovelluksen päivitys Angular-sovellukseksi on monimutkainen prosessi, johon vaaditaan kokemusta molemmista sovelluskehyskiistä. Tätä prosessia voidaan kuitenkin valmistella jo etukäteen päivittämällä olemassa olevaa sovellusta nykyaikaisemmaksi ja soveltaa siihen parhaita kehityskäytänteitä. Angularin kehittäjäjoukko on julkaissut erilaisia työkaluja ja ohjeistuksia helpottamaan päivitystä sovelluskehysten välillä. Työkalun avulla voidaan esimerkiksi analysoida AngularJS-sovelluksen muutostarpeet ennen kuin sovellus on mahdollista päivittää.</p>		
Avainsanat (asiasanat) Angular, AngularJS, päivitys, sovelluskehyski		
Muut tiedot (Salassa pidettävät liitteet)		

Author(s) Lindén, Lasse	Type of publication Bachelor's thesis	Date September 2019
		Language of publication: Finnish
	Number of pages 49	Permission for web publication: x
Title of publication Updating AngularJS application to Angular application		
Degree programme Business Information Technology		
Supervisor(s) Tuikka, Tommi		
Assigned by Protacon Solutions Oy		
<p>Abstract</p> <p>Modern front-end application development is evolving fast. Different application frameworks are often used as a help, since they offer built-in tools for developers. As front-end development is evolving rapidly, these frameworks also become outdated quite fast. Angular is a rewritten framework, and it is the successor of popular AngularJS framework. Angular is rewritten as it needed to keep up with the needs of modern applications by removing weak parts of its predecessor.</p> <p>The purpose of the thesis was to find an as easy as possible way to upgrade existing AngularJS application to modern Angular application. The used research method was development research as getting practically involved in an upgrade process made it possible to gather important information from different parts of the upgrade process. To support the research, theoretical information about these application frameworks and their differences was gathered. Mostly web-based sources were used for theoretical information.</p> <p>The study resulted in a step-by-step guide for how to upgrade the researched application to a modern Angular application. Along with the guide, important information about the differences of these frameworks was obtained. This information can be used as a tutorial when moving from AngularJS project to a modern Angular project.</p> <p>Updating AngularJS application to a modern Angular application is a complicated process that requires experience from both frameworks. The preparation for this process can be started in advance by updating the existing application to be more modern and to use best practices for development. Angular's development team has released different tools and guides to make the updating process easier. The tool can be used to analyze AngularJS applications for parts that need to be changed before the application is ready for update.</p>		
Keywords/tags (subjects) Angular, AngularJS, updating, application frameworks		
Miscellaneous (Confidential information)		

Sisältö

Käsitteet	3
1 Johdanto	4
2 Tutkimusasetelma	5
2.1 Opinnäytetyön toimeksiantaja ja tavoitteet	5
2.2 Tutkimusongelma	6
2.3 Tutkimusmenetelmät	7
3 AngularJS	7
3.1 AngularJS-arkkitehtuuri	8
3.2 Parhaat kehityskäytänteet	12
4 Angular	13
4.1 Angular-arkkitehtuuri	14
4.2 Angular CLI	18
4.3 Parhaat kehityskäytänteet	18
5 Esimerkkisovelluksen päivitys	19
5.1 Valmistelu	19
5.2 Angularin asennus	21
5.3 Päivitä palvelut	23
5.4 Päivitä komponentit	23
5.5 AngularJS poistaminen	27
6 Tutkimustulokset	28
6.1 Tutkimuskohteen esittely	28
6.2 Tutkimuskohteen päivitys	29
6.2.1 Valmistelu	30
6.2.2 Angularin asennus	33
6.2.3 Palveluiden päivitys	35
6.2.4 Komponenttien päivitys	36
6.2.5 Jälkityöt	39

7	Johtopäätökset.....	40
8	Pohdinta.....	42
	Lähteet	45
	Liitteet.....	47
	Liite 1. Step-by-step ohje AngularJS => Angular	47

Kuviot

Kuvio 1. ng-repeat-direktiivin käyttö.....	10
Kuvio 2. Esimerkki tsconfig.json-tiedostosta	20
Kuvio 3. Esimerkki Angularin main.ts-tiedostosta	22
Kuvio 4. Sovelluksen alustus hybridisovelluksena	23
Kuvio 5. Esimerkkisovelluksen AngularJS-komponentti	24
Kuvio 6. Esimerkkisovelluksen päivitetty AngularJS-yhteensopiva Angular- komponentti	25
Kuvio 7. Esimerkkisovelluksen päivitetyn komponentin HTML-mallipohja	26
Kuvio 8. Esimerkki Angular-reititysmoduulista.....	27
Kuvio 9. Webpackin konfigurointi	31
Kuvio 10. Angular asennukseen liittyvät paketit	34
Kuvio 11. Oletus bootstrap-metodin korvaus hybridiversiolla.....	34
Kuvio 12. Angular-palvelun alentaminen AngularJS-yhteensopivaksi	35
Kuvio 13. Ylennetty kolmannen osapuolen AngularJS-moduuli.....	36
Kuvio 14. Väliaikaisesti ylennetty ui-router-tilanhallinta	38

Käsitteet

Angular CLI (Command Line Interface): Komentorivityökalu, jonka avulla voidaan luoda uusi Angular-sovellus ja lisätä uusia sovelluksen osia komentoriviltä.

Angular: Googlen TypeScriptillä kehittämän sovelluskehityksen nimitys versiolle 2 ja siitä eteenpäin.

AngularJS: Googlen JavaScriptillä kehittämän sovelluskehityksen nimitys versiolle 1.

DOM: Dokumenttioliomalli (engl. document object model), joka kuvaa HTML-sivun rakenteen puumallisesti. Mahdollistaa esimerkiksi eri olioiden manipuloinnin.

ES5 (ECMAScript 5): Nimitys JavaScriptin standardin versiolle, joka on yhteensopiva myös vanhempien selaimien kanssa.

Front-end: Nimitys sovelluksen osasta, johon kuuluu esimerkiksi käyttöliittymä.

JavaScript: Selaimessa toimiva ohjelmointikieli, jota käytetään web-sovelluksen toiminnallisuuden lisäämiseen.

npm (Node Package Manager): Node-pakettien hallintaan tarkoitettu työkalu.

Sovelluskehys (engl. application framework): Valitulla ohjelmointikielellä kirjoitettu kehys, jonka päälle rakennetaan oma sovellus hyödyntäen valmiita työkaluja.

TypeScript: JavaScriptin ylijoukko, joka käännetään lopuksi JavaScriptiksi. Mahdollistaa kehityksen aikaisen tarkemman virheiden havaitsemisen tyyppityksen avulla.

Yksisivuinen sovellus (engl. single page application): Sovellus, jossa on vain yksi näkymä. Sivulla liikkuminen tapahtuu näkymää päivittämällä.

1 Johdanto

Nykyaikainen front-end-sovelluskehitys on kehittynyt todella nopeasti. Tämän takia sovelluskehittäjät joutuvat jatkuvasti opiskelemaan uusia asioita pysyäkseen kehityksen mukana. Sovelluskehityksen apuna käytetään usein erilaisia sovelluskehelyksiä, joiden tarkoituksena on tarjota kehittäjille valmiita työkaluja sovelluksen kehittämiseen. Nämä sovelluskehelykset vanhenevat varsin nopeasti, jolloin myös päivityksiä tarvitaan usein. (Ahmed 2018.)

AngularJS on Googlen kehittämä avoimen lähdekoodin sovelluskehys, jonka ensimmäinen versio on julkaistu vuonna 2011. Todella moni asia on muuttunut tuosta hetkestä tähän päivään tultaessa front-end kehityksessä. Vuonna 2016 julkaistun version 2 kohdalla tämä sovelluskehys kirjoitettiin alusta alkaen uudelleen, sillä tämä mahdollisti AngularJS:n heikkojen osien jättämisen kokonaan pois. Versiosta 2 eteenpäin tämä sovelluskehys tunnetaan nimellä Angular ja se on kirjoitettu TypeScriptillä. Tämä mahdollistaa JavaScriptin uudempien versioiden ominaisuuksien käyttämisen yhteensopivana vanhempien versioiden kanssa. (Hevery & Green 2014.)

AngularJS-sovelluksen päivittäminen Angular-sovellukseksi on monimutkainen prosessi. Vanhaa koodipohjaa voidaan muokata parhaiden kehityskäytänteiden mukaiseksi ennen varsinaista sovelluskehelyksen päivitystä. Tämän tarkoituksena on helpottaa tulevaa prosessia. Sovelluksen koko vaikuttaa huomattavasti tämän päivityksen monimutkaisuuteen. (Upgrading from AngularJS to Angular n.d.)

Angularin kehittäjäjoukko on käyttänyt aikaa työkalujen ja ohjeistuksien kehittämiseen, joiden tarkoituksena on tehdä tästä päivityksestä helpompaa. Työkalun avulla voidaan analysoida AngularJS-sovelluksen rakenne ja sovelluksen osat, jotka täytyy muuttaa ennen päivitysprosessin aloitusta. Angulariin on mahdollista lisätä paketti, joka mahdollistaa esimerkiksi sovelluksen eri osien alentamisen AngularJS-yhteensopivaksi. Tämän avulla sovelluksessa voidaan väliaikaisesti käyttää molempia sovelluskehelyksiä samanaikaisesti. (Olson 2018.)

2 Tutkimusasetelma

2.1 Opinnäytetyön toimeksiantaja ja tavoitteet

Tämän opinnäytetyön tavoitteena on tutkia, miten olemassa oleva AngularJS-sovellus saadaan mahdollisimman helposti päivitettyä käyttämään uudempaa versiota Angular-sovelluskehyksestä. Toimeksiantajana tässä opinnäytetyössä toimii ohjelmistoyritys Protacon Solutions Oy, jonka pääpaikkana toimii Jyväskylä.

AngularJS-sovelluksia on vielä paljon, mutta uudempi, täysin uudelleenkirjoitettu Angular on nykyisin sovelluskehys, joka todennäköisemmin valitaan uudelle **web-sovellus**-projektille pohjaksi **front-end** puolen kehitykseen. Nykyisin suositellaankin ennen päivitystä valmistelevaan vanhempi AngularJS-sovellus muistuttamaan uudemman version tyyliä. Tähän on tarjolla avuksi kehittäjäjoukon suosituksia parhaista käytänteistä, joiden avulla päivityksen tekeminen onnistuu helpommin. (Upgrading from AngularJS to Angular n.d.) Angular ei ole ainut vaihtoehto mihin AngularJS-sovellus voidaan päivittää, vaan tarjolla on myös monia muita tunnettuja vaihtoehtoja, esimerkiksi Vue tai React. Tämä tutkimus keskittyy kuitenkin tutkimaan AngularJS-sovelluksen päivittämistä Angular-sovellukseksi, joten nämä muut vaihtoehdot ovat rajattu pois tästä tutkimuksesta.

Aiheen ajankohtaisuudesta kertoo Googlen ilmoitus tammikuussa 2018, jossa kerrottiin kehittäjäjoukon valmistelevan viimeistä suurempaa päivitystä AngularJS:lle. Kyseinen päivitys on julkaistu heinäkuussa 2018. Sovelluskehys on kirjoitushetkellä kolmen vuoden Long Term Support-ajanjaksolla heinäkuusta 2018 alkaen. Tämä tarkoittaa käytännössä, että uusia ominaisuuksia ei enää kehitetä. Kehittäjäjoukon tarjoama tuki loppuu tämän ajanjakson jälkeen vuonna 2021. Tämän kolmen vuoden aikana joukko kehittäjiä keskittyy korjaamaan vain vakavia bugeja. Vakava bugi voi tässä tapauksessa tarkoittaa esimerkiksi jonkin suurimman selaimen tuen loppumista AngularJS-versiolle 1.7.x. Myös saman version havaitut tietoturva-aukot ovat tuen piirissä ja ne korjataan Long Term Support-ajanjaksolla. (Darwin 2018.)

AngularJS-sovelluksen päivittäminen Angular-sovellukseksi on ollut myös vahvasti esillä Angular-kehittäjille tarkoitettussa konferenssitapahtumassa nimeltä ng-conf viime vuosina. Tapahtumassa on ollut monia esityksiä liittyen päivitykseen ja prosesseihin, jotka tekevät päivittämisestä helpompaa.

2.2 Tutkimusongelma

Tutkimusprosessi koostuu yleensä viidestä eri vaiheesta. Se aloitetaan ideatasolla, johon kuuluu esimerkiksi tutkimuksen tavoitteen suunnittelu, tutkimuksen aikataulun suunnittelu, sekä tutkimusongelman hahmottaminen. Ideatason jälkeen vuorossa on sitoutuminen tutkimukseen. Tämän vaiheen aikana tehdään tutkimussuunnitelma, jossa käsitellään tarkemmin ideatasolla esille tulleita asioita ja hankitaan tutkimuksen toteuttamiseen liittyvät luvat ja sopimukset. Näiden kahden vaiheen jälkeen voidaan aloittaa tutkimuksen toteuttamisvaihe, jonka aikana hankitaan tutkimusaineistoa ja tulkitaan hankittua tietoa. Kun tutkimus on toteutettu ja tuloksia voidaan analysoida, aloitetaan kirjoittamisvaihe. Kirjoittamisvaihe on käytännössä koko ajan tutkimuksen aikana mukana, vaikka se voi tuntua yhdeltä selkeältä osalta tutkimusta. Kirjoittamisvaiheeseen kuuluu esimerkiksi toteuttamisvaiheen aikana kirjoitetut muistiinpanot. Tutkimuksen viimeisenä vaiheena toimii tiedottaminen tutkimuksen tuloksista. Tässä vaiheessa pidetään yleensä tiedotustilaisuus ja arkistoidaan tutkimus ja käytetyt tutkimusaineistot. (Vilkka 2015.)

Vilkan (2015) mukaan ideointivaiheessa on tärkeää käyttää riittävästi aikaa tutkimusongelman ja siihen liittyvien tutkimuskysymysten huolelliseen määrittelyyn, sillä huonosti rajattu tutkimusongelma voi johtaa merkityksettömiin tutkimustuloksiin. Tutkimusongelman määrittelyssä käytetään apuna tutkimuskysymyksiä. Tutkimuskysymykset koostuvat yleensä yhdestä pääkysymyksestä, jonka tarkoituksena on määrittellä suunta tutkimusongelmalle. Muista kysymyksistä käytetään nimitystä teoreettiset tutkimuskysymykset, eli nämä ovat kysymyksiä joihin tutkimuksessa halutaan vastata.

Tämän opinnäytetyön tutkimusongelmana on kehittää toimeksiantajayritykselle helppo tapa päivittää olemassa oleva AngularJS-sovellus Angular-sovellukseksi. Tutkimusongelmasta johdettuina tutkimuskysymyksinä toimivat seuraavat tutkimuskysymykset:

- Mitä AngularJS-sovelluksen päivittämiseen Angular-sovellukseksi tarvitaan?
- Kuinka suuri työpanos ja osaaminen päivittämiseen tarvitaan?
- Miten päivittämisestä voi tehdä mahdollisimman helppoa

2.3 Tutkimusmenetelmät

Tämän opinnäytetyön aihe on käytännönläheinen. Tutkimuksen tarkoituksena on parantaa olemassa olevaa web-sovellusta päivittämällä sen front-endin pohjana toimiva sovelluskehys uudempaan versioon. Tähän tutkimukseen menetelmäksi soveltuu parhaiten kehittämistutkimus, sillä tutkimuksen tavoitteena on ratkaista käytännön ongelma. Kehittämistutkimus on aina liitoksissa käytännön työhön. Poikkeuksena tähän on puhtaasti teoreettiset tutkimukset esimerkiksi matematiikkaan ja fysiikkaan liittyen. (Kananen 2012, 13.)

Kanasen (2012, 43) mukaan kehittämistutkimus on kehittämistyötä, joka täyttää tutkimuksen kriteerit. Tämän tyyppisen tutkimuksen vaarana on tutkimusotteen puuttuminen, eli kehittämistyön kohteena on yleisiä asioita ilman tieteellistä otetta. Kehittämistutkimuksessa ei pyritä yleistämään ongelmaa, vaan tuloksena on kehittämisen kohteena olleen ilmiön muutos entiseen verrattuna.

3 AngularJS

AngularJS on Googlen kehittäjäjoukon kehittämä avoimen lähdekoodin sovelluskehys yksisivuisten web-sovellusten front-end puolen kehitykseen. Angular on alun perin Googlen työntekijän nimeltä Miško Hevery sivuprojekti, joka tunnettiin aluksi nimellä Get Angular. Sen tarkoituksena oli vähentää manuaalista työtä, joka tarvittiin web-pohjaisen sovelluksen kehittämiseen. Kun web-sovellusta lähdetään rakentamaan

täysin nollasta ilman sovelluskehystä, vaaditaan paljon pohjatyötä ennen kuin varsinaista front-end-puolta voidaan lähteä kehittämään. Esimerkiksi yhteydet tietokantaan ja turvallisuuteen liittyvät vaiheet voivat viedä paljon aikaa ja vaivaa. Angularin alkuperäinen käyttötarkoitus oli nimenomaan tietokannan hallinta sekä turvallisuuden parantaminen web-sovelluksessa ja mahdollisuus upottaa web-sovellus näkymään toisella verkkosivulla. Käytössä on rakenteelle HTML, tyylille CSS ja logiikalle JavaScript, joten uusia web-sovelluksen luomiseen tarvittavia kieliä ei tarvitse opetella. Tarkoituksena olikin alun perin luoda työkalu web-suunnittelijoille, joilla ei ole paljoa kokemusta ohjelmoinnista. Ja koska käytössä on HTML ja CSS, on ulkoasun muokkaaminen samanlaista kuin ilman sovelluskehystä. (Hevery & Green 2014.)

Nimitys Angular tulee Heveryn ja Greenin (2014) mukaan HTML-dokumenteissa käytetyistä kulmikkaista sulkumerkeistä, esimerkiksi ”<div></div>”. Angularin logo on myös ottanut vahvasti vaikutteita HTML 5-logosta. Sovelluskehystä käytettiin aluksi Googlen omien palveluiden kehittämiseen. Aluksi ylempi johto ei nähnyt Angularin mahdollisuuksia menestykselle, mutta tiimi jatkoi tästä huolimatta kehitystyötä ja lopulta muuttivat lähdekoodin kaikille avoimeksi. Tämän seurauksena kehitys nopeutui ja tavoitteena olikin julkaista vakaa versio elokuussa 2010. Lopulta yhdeksän kuukautta myöhemmin toukokuussa 2011 julkaistiin AngularJS-versio 1.0. Tammikuussa 2013 suosio lähti suureen nousuun ja nousi hetkessä selvästi kilpailijoiden ohi. (Hevery & Green 2014.)

3.1 AngularJS-arkkitehtuuri

Arkkitehtuurisesta näkökulmasta pääkonseptina on MVC-arkkitehtuurimalli. Tämän mallin mukaan sovellus koostuu kolmesta eri palasesta. **Malli** (engl. model) tarkoittaa sovelluksen taustalla olevaa rakennetta, eli käytännössä ohjeistusta kuinka tietoa haetaan palvelimelta ja mihin haettu tieto sijoitetaan sovelluksessa. **Näkymä** (engl. view) viittaa nimensä mukaisesti sovelluksen käyttäjälle näkyvään osuuteen, eli sovelluksen käyttöliittymään, joka muodostetaan määritetyn mallin mukaisesti. **Käsittelijä** (engl. controller) vastaa sovelluksen logiikasta ja sen avulla haetaan esimerkiksi tietoa palvelimelta ja määritetään mitä osia siitä näytetään. Käsittelijäkerroksesta

voidaan käyttää myös nimitystä **näkymämalli** (engl. viewmodel). Tämän arkkitehtuurivalinnan ansiosta koodia on helpompi ylläpitää ja käyttää uudelleen, sillä sovelluksen eri kerrokset eivät ole suoraan riippuvaisia toisistaan. (Seshadri & Green 2014, 2–3.)

Moduuli

Moduulit ovat kuin paketteja, jotka sisältävät koodia. Moduulien tarkoituksena AngularJS-sovelluksessa on liittää koodit moduuleihin, jotta ne voidaan aktivoida bootstrap-metodin avulla sovellukseksi. Jokaiseen moduuliin voidaan määrittää omat käsittelijät, palvelut sekä direktiivit. Esimerkiksi moduuliin liitetyt käsittelijät voivat käyttää saman moduulin palveluita tiedon hakemiseen. Moduuleihin voidaan määrittellä riippuvuuksia myös toisiin moduuleihin, jolloin myös kyseisen moduulin funktioita voidaan käyttää. (Seshadri & Green 2014, 15.)

Omien moduulien luomisessa käytetään `angular.module('myApp', []);`. Luontivaiheessa toiseksi parametriksi asetetaan riippuvuudet taulukkomuodossa. Jos toista parametria ei aseteta, yrittää AngularJS hakea olemassa olevaa moduulia ja aiheuttaa virheen, jos sitä ei löydy. Moduulien käyttö mahdollistaa nopeamman yksikkötestauksen, sillä testausta varten ei tarvitse ladata kaikkia moduuleja kerrallaan. Testaukseen riittää vain sen moduulin haku, johon testattava koodi kuuluu.

Käsittelijä

Käsittelijät (engl. controller) ovat suuressa roolissa datan käsittelyssä AngularJS:n aikaisemmissa versioissa. Käsittelijät määrittelevät moduuleittain, esimerkiksi `angular.module('myApp').controller('MainCtrl', [function() {...}]);`. Tässä tilanteessa haetaan ensin moduuli `myApp` ja liitetään sen osaksi uusi käsittelijä. Käsittelijöiden avulla voidaan esimerkiksi päättää mitä käyttöliittymässä näytetään. Käsittelijöiden avulla saadaan myös haettua tietoa palveluista. Käsittelijät ovat tiukasti sidoksissa käyttöliittymään, eikä sovelluksessa tulisi koskaan käyttää käsittelijää ilman sen sitomista käyttöliittymään. Jos osana käsittelijän tehtäviä on logiikkaa, joka ei liity millään tavalla käyttöliittymään, tulisi tämä osuus siirtää sille sopivaan palveluun. AngularJS-versiossa ennen 1.2 on injektoitu `$scope`-muuttuja käsittelijälle. Tämä on mahdollistanut muuttujien määrittelyn vain kyseiselle käsittelijälle. Tämä on myöhemmin

kuitenkin korvattu controllerAs-syntaksilla myöhemmissä versioissa. Tämä mahdollistaa muuttujien määrittelyn this-avainsanalla, jolloin \$scopea ei enää tarvita. Määriteltäessä muuttujia käyttäen avainsanaa this, suositellaan ensin määrittämään tämä eri nimiseen muuttujaan esimerkiksi "var self = this;", jotta sovelluksen kehittäjät tietävät tarkalleen mihin uutta muuttujaa ollaan määrittämässä. Avainsana this on ongelmallinen, koska funktion sisältä ja ulkoa kutsuttuna tämä avainsana voi viitata täysin eri muuttujaan. (Seshadri & Green 2014, 17–22.)

Direktiivi

Direktiivit tarkoittavat ohjeistuksia HTML-kääntäjälle, miten määritettyä elementtiä tulisi käsitellä käännösvaiheessa ja mikä toiminnallisuus siihen liitetään. Käsittelijän avulla voidaan käyttää myös monia valmiita AngularJS-direktiivejä. Nämä direktiivit alkavat aina ng-etuliitteellä. Valmiiden direktiivien avulla voidaan esimerkiksi määritellä ehto, milloin kyseinen elementti näytetään käyttöliittymässä tai määrittää ehdollinen luokka elementille. Tämän lisäksi on myös mahdollista luoda omia direktiivejä esimerkiksi tapahtumakuuntelijoiden lisäämiseksi elementille tai asettaa päivämäärän näyttämiseksi haluttu muoto. (Creating Custom Directives n.d.)

Direktiivi ng-repeat on direktiivi, jonka avulla voidaan esimerkiksi luoda useita HTML-elementtejä käsittelijällä saatavilla olevasta taulukosta. (ks. kuvio 1) Tämän direktiivin käyttäytyminen on samanlainen kuin "for each"-silmukka useissa ohjelmointikielessä. Tätä direktiiviä käytetään AngularJS-sovelluksissa erittäin paljon, sillä se antaa mahdollisuuden käydä läpi sille annetun taulukon ja luoda näkymään elementtejä tarpeen mukaan. Tämän lisäksi elementille voidaan määrittää erilaisia filttareita, joita tulee AngularJS:n mukana. Paljon käytetty filtti taulukoissa on esimerkiksi orderBy, jonka avulla voidaan järjestää taulukon tiedot. (Seshadri & Green 2014, 22–24.)

```
<ul>
  <li ng-repeat="note in $ctrl.notes">
    <h1>{{note.label}}</h1>
    <p>Done: {{note.done}}</p>
  </li>
</ul>
```

Kuvio 1. ng-repeat-direktiivin käyttö

Palvelu

Direktiivien lisäksi AngularJS tarjoaa valmiiksi sisäänrakennettuna erilaisia palveluita. Palvelut ovat funktioita tai objekteja, joiden tarkoituksena on säilyttää sovelluksen toistuvaa käyttäytymistä sekä tilaa. Tämä tarkoittaa esimerkiksi sitä, että jokin taulukko päivitetään ja haetaan usealla käsittelijällä. Paljon käytetty valmis `$http`-palvelu on tehty helpottamaan http-kutsujen kanssa, sillä se on yleisesti ja yleensä koko sovelluksen laajuisesti käytetty palvelu. Sen avulla voidaan hakea määritellystä osoitteesta tietoja esimerkiksi käsittelijälle tai toiselle palvelulle. Http-palvelun kutsut palauttavat lupauksen (engl. promise). Lupauksen avulla voidaan ketjuttaa kutsuja ja määrittää mitä sovelluksessa tapahtuu, kun http-kutsulle saadaan vastaus (Poshtaruk 2019). Sisäänrakennettujen palvelujen lisäksi voidaan luoda myös omia palveluita tiettyyn tarkoitukseen. Palvelut soveltuvat erityisen hyvin tiedon säilömiseen, jotta sitä voidaan käyttää useassa sovelluksen osassa. Ja koska palvelut luodaan vain kerran ja säilytetään sovelluksen ajan ajan, on palvelun tiedot ja funktiot saatavilla tänä aikana ilman luontia uudelleen. Jos palvelua halutaan käyttää, tulee riippuvuus palvelulle injektoida luontivaiheessa. (Seshadri & Green 2014, 69–78.)

Komponentti

Versiosta 1.5 eteenpäin AngularJS-sovelluksiin on ollut mahdollista ottaa käyttöön uusi komponenttirajapinta, jonka avulla voidaan luoda omia HTML-elementtejä. Tämä on myös aikaisemmin ollut mahdollista direktiivien avulla. Komponenttirakenteen avulla pakotetaan hyvien käytänteiden soveltaminen aiempaan direktiivimalliin verrattuna. Komponenttien avulla sovelluksen rakennetta muokataan muistuttamaan uudempaa Angularia ja erillisten komponenttien yksikkötestaus on paljon helpompaa. Uudelleenkäytettävien komponenttien on tarkoitus korvata aiemmin käytetyt käsittelijät osittain. Uudella komponenttirakenteella suositellaan käytettävän yksisuuntaista datasidontaa (engl. one-way data binding). Komponenttien kanssa suositellaan käytettävän tapahtumakäsittelijöitä kaksisuuntaisen datasidonnan tilalla, sillä se on tehokkaampi tapa kommunikoida sivun kanssa ja silloin rakenne muistuttaa myös enemmän uudempaa Angularia. (Wilken 2016.)

Reititys

AngularJS on yksisivuisten sovellusten kehittämiseen tarkoitettu sovelluskehys. Sovelluksen navigointi eri näkymien välillä tapahtuu hyödyntäen sovelluskehiksen tarjoamaa reititintä. Reitittimen tarkoituksena on ylläpitää selaimen tilaa, jolloin edelliseen näkymään voidaan palata selaimen historian perusteella. AngularJS tarjoaa ngRoute-nimisen moduulin, jonka avulla on mahdollista luoda reititys sovellukseen. Tämä moduuli on aiemmin ollut suoraan osa AngularJS-ydintä, mutta rinnalle nousseiden avoimen lähdekoodin reititysmahdollisuuksien seurauksena tämä tarvitsee ottaa erikseen käyttöön. (Seshadri & Green 2014, 139–141.)

Perinteiseen web-sovellukseen verrattuna yksisivuisessa sovelluksessa ei päivitetä koko sivua uudelleen navigointivaiheessa, vaan päivitys voidaan rajoittaa haluttuun osaan sovelluksen näkymää. Perinteisessä web-sovelluksessa voidaan navigoida näkyvän sivun tiettyyn osioon, mutta AngularJS mahdollistaa samantyyllisellä #-osoitteen määrittelyllä näkymän päivittämisen toiseen ilman liikkumista sivulta toiselle. (Seshadri & Green 2014, 140.) Wilkenin (2016) mukaan AngularJS-versiossa 1.5 esitelty komponenttirakenne mahdollistaa reitittimen osoittamisen suoraan komponenttiin.

3.2 Parhaat kehityskäytänteet

Tyylioppaat sisältävät parhaita käytänteitä ja hyvät perustelut miksi kyseisiä käytänteitä kannattaa käyttää AngularJS-sovelluksen kannalta. Oppaiden tarkoituksena on parantaa sovelluksen rakennetta ja tehdä koodista helppolukuisempaa. (Papa n.d.) Mahdollisten virhetilanteiden varalta on myös helpompaa, jos sovellus noudattaa tyyliopasta. Tällöin ratkaisun löytäminen helpottuu, sillä myös muilla sovelluksilla on todennäköisemmin samanlainen rakenne ja ongelma. Blackin (2014) mukaan tyylioppaita on useiden eri henkilöiden kirjoittamana, mutta John Papan kirjoittama opas on kaikkein tunnetuin ja ajantasaisin. Sisällöltään eri tyylioppaat ovat melko samanlaisia, mutta niiden rakenne ja laajuus eroaa toisistaan. Tyylioppaat sisältävät hyviä käytänteitä esimerkiksi nimeämiseen ja sovelluksen rakenteeseen liittyen.

4 Angular

Angular on uudelleenkirjoitettu sovelluskehys ja seuraaja suositulle AngularJS-sovelluskehyselle. Heveryn ja Greenin (2014) mukaan AngularJS on vanhempien selaimien sovelluskehys ja uusi Angular, versiosta 2 eteenpäin on uudempien selaimien sovelluskehys. Angularin versioiden 1 ja 2 välissä on Angular Dart, joka on Dart-ohjelmointikielellä kirjoitettu sovelluskehys. Angular Dart on uudelleenkirjoitettu ja siihen on otettu mukaan AngularJS:n parhaita käytänteitä. Dart-version mukana on paljon ominaisuuksia, jotka on myöhemmin otettu myös osaksi Angularia versiosta 2 eteenpäin.

Angular on kirjoitettu mobiilikehitys edellä, eli kehittäjäjoukko on panostanut sovelluksien toimintaan myös mobiililaitteilla. Aikaisemmin tämä on toteutettu erillisellä sovelluskehysellä. Angularin ollessa avoimen lähdekoodin sovelluskehys, myös yhteisö on ajanut eteenpäin kehitystä mobiililaitteille. Mobiililaitteiden tuen lisäksi kehittäjäjoukko on panostanut myös sovelluksien toimimiseen ilman verkkoyhteyttä. Zone.js esiteltiin Dartin ominaisuutena ja se on otettu mukaan myös Angulariin. Tämän avulla on mahdollista suorittaa koodia säikeissä ja säilyttää sovelluksen aikaisemmat tilat muistissa. Tämä mahdollistaa esimerkiksi poikkeuksien tarkemman tarkastelun, sillä tämän avulla on mahdollista saada tarkempaa tietoa poikkeuksen aiheuttajasta. (Hevery & Green 2014.)

TypeScript

TypeScript on Microsoftin kehittämä avoimen lähdekoodin JavaScriptin ylijoukko, joka on kehitetty laajentamaan JavaScriptin ominaisuuksia. Se mahdollistaa monia kehityksen aikaisia hyötyjä, kuten valinnaisen vahvan tyyppityksen ja uudempien ES-versioiden natiivien ominaisuuksien käyttämisen. Esimerkiksi uusia ominaisuuksia voidaan käyttää ES5-yhteensopivana kääntäjän avulla, jolloin uudet ominaisuudet toimivat myös vanhemmilla selaimilla. (TypeScript – JavaScript that scales n.d.) Papan ja Wahlinin (2017) mukaan TypeScriptin konfiguraatiossa voidaan määritellä mihin ES-versioon TypeScript käännetään ja määritellä myös useita muita kääntäjän asetuksia.

Angular versiosta 2 eteenpäin on kirjoitettu TypeScriptillä. Tämä mahdollistaa esimerkiksi luokkien käytön, joka on tärkeä ominaisuus Angularissa. Tämän lisäksi on mahdollista hyödyntää myös liittymiä (engl. interface) määrittämään omia mukautettuja tyyppityksiä. Mukautetut palautustyyppit mahdollistavat kehityksenaikaisen virheiden havaitsemisen. Jos palautustyyppille on määritelty liittymä, näyttää TypeScript virheen, jos tämän kirjoitusasu ei vastaa liittymää. Toinen erityisen tärkeä ominaisuus TypeScriptissä on moduulit, jotka mahdollistavat tarvittavien koodien hakemisen vain yhden tiedoston käyttöön, jossa niitä tarvitaan. Näitä moduuleita voidaan ladata esimerkiksi SystemJS- tai Webpack-moduulienlataajien avulla sovelluksen käyttöön. (Papa & Wahlin 2017.) Tässä tärkeää on huomioida, että Angularin moduulit ja TypeScriptin moduulit eivät ole yhteyksissä toisiinsa (Introduction to modules n.d.).

4.1 Angular-arkkitehtuuri

Angularin arkkitehtuurissa on yhtäläisyyksiä edeltäjäänsä AngularJS verrattuna, mutta myös paljon eroavaisuuksia. Angulariin on otettu mukaan AngularJS:n parhaita käytänteitä, jotka ovat hyviä esimerkiksi suorituskyvyltään.

Angular-moduulit

Angularin moduulit ovat erittäin tärkeä osa Angular-sovellusta. Moduulien avulla voidaan jakaa sovellus loogisiin kokonaisuuksiin, joita voidaan helposti testata yksikkötestien avulla. Angularin mukana tulee valmiita moduuleita eri tarkoituksiin. Näistä esimerkkinä ”BrowserModule”, jota käytetään, kun halutaan suorittaa sovellusta verkkoselaimessa. (Frequently Used Modules n.d.)

Angulariin itse luodut moduulit määritellään aina @NgModule-määrittelyllä, jolloin moduulin metatietoihin on mahdollista määritellä mitkä osat sovelluksesta kuuluvat kyseiseen moduuliin. Tärkein osa jokaista Angular-sovellusta on juurimoduuli, joka on hyvien käytänteiden mukaisesti nimetty AppModuleksi. Tämän moduulin tarkoituksena on hoitaa varsinainen logiikka sovelluksen näyttämiseen verkkoselaimessa Angularin bootstrap-metodia hyödyntäen. Hyvien käytänteiden mukaisesti sovelluksen käynnistäminen tapahtuu AppComponent-nimisessä komponentissa. (Introduction to modules n.d.)

Angular moduulin "imports"-tietoihin määritellään taulukkomuodossa kyseisen moduulin tarvitsemat toiset moduulit. Eli jos moduulin palveluissa käytetään esimerkiksi Http-moduulin kutsuja palvelimen kanssa keskusteluun, tarvitsee HttpClientModule määritellä "imports"-kohdassa. Toisten moduulien lisäksi Angular-moduulille voidaan määritellä palveluita, jotka luetellaan taulukkomuodossa "providers"-kohdassa. Toisten moduulien ja palveluiden lisäksi Angular-moduuleille voidaan luoda komponentteja ja direktiivejä. Nämä määritellään moduulin "declarations"-kohdassa. Jos moduulin osia tarvitaan myös muiden moduulien komponenttien käyttöön, määritellään ne "exports"-kohdassa. (Introduction to modules n.d.)

Komponentit

Komponentit ovat vastuussa Angularin näkymästä. Angular-komponentit määritellään @Component-määrittelyllä. @Component-määrittely mahdollistaa valitsijan (engl. selector), mallipohjan (engl. template) ja komponenttikohtaisten tyylien määrittelyn. Valitsijaa käytetään reitityksen yhteydessä tai komponentin näyttämiseen sen ollessa toisen komponentin alikomponentti. Komponenttiin liittyvä mallipohja voidaan määritellä joko suoraan metatietoihin backtickien avulla template-määrittelyllä. Toinen tapa luoda komponentin mallipohja on luoda erillinen HTML-tiedosto, joka haetaan templateUrl-määrittelyn avulla relaatiivisesti samasta moduulista. (Introduction to components n.d.)

Kun komponentin metatiedot on määritetty, luodaan varsinainen komponentin loogiikka luokkarakenteen avulla. Komponentin käyttö muualla sovelluksessa määritellään käyttämällä luokkamäärittelyssä "export"-avainsanaa. Tällöin komponentti voidaan liittää myös moduulien "declarations"-listaukseen. Komponenttien mallipohjissa käytetään normaaleja sulkeita tapahtumakäsittelijöiden luomiseksi ja hakasulkeita arvojen määrittelyyn. Esimerkiksi (click)="exampleEvent()" -määrittelyllä luodaan elementille tapahtumakuuntelija, joka aktivoi komponentin metodin tapahtuman toteutuessa. Mallipohjassa esiintyville elementeille voidaan asettaa arvoja hakasulkeiden avulla, esimerkiksi . Jos komponentilla olevia arvoja halutaan näyttää mallipohjassa, onnistuu se käyttämällä kaksinkertaisia aaltosulkeita elementin tekstikohdassa esimerkiksi <div>{{exampleValue}}</div>. (Introduction to components n.d.)

Putket

Angular tarjoaa sisäänrakennettuna valmiina putkia (engl. pipes), joiden avulla voidaan muokata komponentin mallipohjan tekstien ulkoasua. Angularin valmiiksi tarjoamien putkien avulla voidaan muokata kirjaimien ulkoasu vain pieniin tai suuriin kirjaimiin. Myös päivämäärien näyttämiseen voidaan määritellä putki `"| date"` ja sille voidaan antaa parametriksi muoto, jossa päivämäärä näytetään `"MM/dd/yy"`-formaattissa. Angularin kehittäjäjoukko on jättänyt tarkoituksella AngularJS:n taulukoihin soveltuvat `"filter"`- ja `"orderBy"`-putket pois uudesta Angularista. Tämä johtuu suorituskyvystä, sillä taulukon järjestäminen ja tiedon filtteröinti reaaliajassa on todella raskasta. Jos näitä kutsuja olisi monia käynnissä samalla kertaa, voisi taulukon järjestyspyyntöä tulla monia kertoja sekunnissa. (Pipes n.d.)

Angularissa on myös mahdollista luoda omia putkia, joiden avulla voidaan luoda mallipohjien käyttöön esimerkiksi numeroille eksponentiaalinen kerroin. Putki luodaan hyödyntämällä `@Pipe`-määrittelyllä, jolloin metatietoihin voidaan antaa parametrit `"name"` ja `"pure"`. `"Name"` tarkoittaa nimeä, jonka avulla putki haetaan mallipohjan käyttöön. Parametrin `"pure"`-arvo voi olla joko tosi tai epätosi. Arvon ollessa tosi putki ajetaan ainoastaan kyseisen elementin arvon muuttuessa. Jos arvo on epätosi, suoritetaan putki jokaisella komponentin muutoksen tarkistus syklillä. Tämä tarkoittaa sitä, että putki suoritetaan todella usein ja saattaa aiheuttaa suorituskyvyn ongelmia. (Pipes n.d.)

Direktiivit

Direktiivejä on olemassa kolmentyyppisiä. Komponentit ovat direktiivejä mallipohjan kanssa, jolloin nämä määritellään `@Component`-määrittelyllä. Muut kaksi direktiiviä eroavat komponenteista, sillä niiden tarkoituksena on vain manipuloida **DOM**iä. Attribuutti-direktiivin tarkoituksena on muokata olemassa olevan elementin tyyliä tai sen käyttäytymistä. Näitä on valmiina Angularin mukana, esimerkiksi `NgStyle` tyylien määrittämiseen. Rakenteellinen direktiivi muokkaa mallipohjan rakennetta esimerkiksi lisäämällä siihen uusia elementtejä. Angularin mukana tulee valmiina useita tämäntyyppisiä direktiivejä, joista käytetyimmät ovat `*ngFor` ja `*ngIf`. `NgFor`-direktiiviä käytetään taulukon tietojen näyttämiseen mallipohjassa muistakin ohjelmointikie-

listä tutun for-silmukan tapaan. NgIf-direktiivillä voidaan määrittää ehto, milloin elementti luodaan käyttöliittymään. Valmiiden direktiivien lisäksi on myös mahdollista luoda omia direktiivejä @Directive-määrittelyllä. (Attribute Directives n.d.)

Palvelut

Angularin palveluita käytetään metodien luomiseen, joita halutaan kutsua useista komponenteista. Esimerkiksi lokitus tapahtuu usein logger-nimisen palvelun avulla ja sitä kutsutaan useista eri komponenteista. Myös logiikkaa varten, joka ei ole suoraan yhteydessä tiettyyn näkymään, voidaan luoda palvelu. Palveluja käytetään paljon tietojen hakemiseen palvelimelta Angularin mukana tulevan http-palvelun avulla. Http-palvelun avulla voidaan lisätä, lukea, päivittää tai poistaa tietoja palvelimelta. Tämä palvelu palauttaa aina tarkkailijan (engl. observable), joka on toiminnaltaan samankaltainen kuin lupaus. Tarkkailijan arvolle voidaan luoda kuuntelijoita, joiden avulla voidaan määrittää mitä tarkkailijalta saadulla arvolla halutaan tehdä. Verrattuna lupaukseen, tarkkailija mahdollistaa esimerkiksi kutsun helpomman perumisen ja uudelleenyrityksen. Palveluiden luomiseksi metatiedot määritellään @Injectable-määritelmän avulla. Tämä mahdollistaa riippuvuusinjektion (engl. dependency injection), jonka avulla palvelua voidaan käyttää toisesta palvelusta tai komponentista. Jotta injektoidut palvelut voidaan käyttää, tarvitsee ne määritellä konstruktorissa. (Introduction to services and dependency injection n.d.)

Reititys

Angular on yksisivuisten sovellusten tekoon suunniteltu sovelluskehys. Sovelluksessa siirtyminen tapahtuu mukana tulevan reitittimen avulla. Tällöin käyttöliittymässä näkyvää näkymää päivitetään sen mukaan mikä reitti on kulloinkin aktivoituna. Reititin toimii samalla periaatteella kuin perinteinenkin verkkosivusto, eli selaimen avulla voidaan mennä takaisinpäin sovelluksessa ja selata historiatietoja. Reititin otetaan käyttöön RouterOutlet-nimisellä elementillä, joka toimii näkymän sijoituspaikkana. Reitittimen konfiguraatiossa määritellään reittejä ja mikä komponentti näkymässä näytetään määritellyn reitin ollessa aktiivisena. Jotta käyttäjä voi liikkua sivulla reitittimen avulla, pitää navigaatio-elementille määritellä routerLink-arvo. Tämä arvo kertoo reitittimelle, minkä reitin halutaan aktivoituvan. Reitittimelle voidaan myös määrittää

oletusarvo, jos pyydetty reitti ei vastaa mitään olemassa olevaa reittiä. Reititin mahdollistaa myös reittien suojaamisen eli tarkistuksen saako käyttäjä nähdä pyydetyn näkymän. Tämä mahdollistaa esimerkiksi sisällön rajoittamisen riittämättömillä käyttöoikeuksilla tai takaisinpäin siirtymisen, jos sivun tietoja ei olla tallennettu. (Routing & Navigation n.d.)

4.2 Angular CLI

Angular-sovelluksien tekemiseen on kehitetty kehittäjäjoukon toimesta komentorivityökalu (engl. command line interface), jonka tarkoituksena on helpottaa sovelluksen luontia tarjoamalla valmis rakenne sovellukselle. Uuden Angular-sovelluksen voi luoda komentoriviltä komennolla `"ng new <projektin-nimi>"`. Komentorivin kautta voidaan myös luoda esimerkiksi uusia komponentteja ja määrittää niille moduulit, johon komento `"ng generate"` luo viittaukset automaattisesti. Komentorivin avulla voidaan myös automatisoida tehtäviä, kuten testien ajo tai paikallisen kehityspalvelimen pystytys. Komennolla `"ng serve"` voidaan rakentaa sovellus ja tarkkailla lähdekoodissa tapahtuvia muutoksia, joihin reagoidaan uudelleenrakentamalla sovellus ja päivitettyt muutokset näkyvät automaattisesti. (CLI Overview and Command Reference n.d.)

4.3 Parhaat kehityskäytännöt

Toisin kuin AngularJS-tyyliopas, Angularin tyyliopas on virallinen osa sen dokumentaatiota. Molemmat tyylioppaat sisältävät samankaltaisia parhaita kehityskäytänteitä. Tiedostojen ja sovelluksen osien nimeäminen tulisi noudattaa rakennetta, jonka avulla voidaan nimen perusteella päätellä mitä kyseinen tiedosto tai luokka sisältää. Luomalla uuden Angular-sovelluksen CLI:n avulla, luodaan sovellukselle myös oletuksena suositeltu kansiorakenne ja hyvien käytänteiden mukaisesti nimetty juuri-moduuli. Komentorivin kautta luoduille sovelluksen osille lisätään myös automaattisesti tyyppi mukaan tiedostonimeen. Tärkein osa tyyliopasta on yhden sääntö eli yhdessä tiedostossa tulisi olla esimerkiksi vain yksi komponentti. Tämän lisäksi kannattaa pyrkiä pitämään koodirivit tiedostossa alle 400. Näiden käytänteiden noudattaminen tekee koodista luettavamman ja helpomman testata. (Style Guide n.d.)

5 Esimerkkisovelluksen päivitys

Tässä opinnäytetyössä tutustutaan ensin esimerkkisovelluksen päivitykseen, jotta saadaan yleiskäsitys mitä päivityksessä kannattaa ottaa huomioon. Esimerkkisovellukseksi tässä opinnäytetyössä valikoitui virallinen AngularJS-tutoriaali. Tämä kyseinen tutoriaali on AngularJS-kehittäjäjoukon kehittämä ohjeistus ensimmäisen AngularJS-sovelluksen tekemiseen. Tutoriaalin aikana hyödynnetään hyviä kehityskäytänteitä ja sovelluksen rakenne on varsin yksinkertainen. Tämä sovellus valikoitui parhaaksi tavaksi tutustua päivitysprosessiin ja sen perusteisiin. Perusteiden opettelu on tässä kohdassa tärkeää, jotta saadaan yleiskäsitys, miten päivitysprosessi etenee. Tämän opinnäytetyön varsinainen tutkimuskohde on kooltaan huomattavasti suurempi ja rakenne paljon monimutkaisempi.

5.1 Valmistelu

Ennen kuin päivitysprosessia aloitetaan, tulisi vanha koodipohja muokata vastaamaan AngularJS-kehittäjäjoukon John Papan kirjoittamaa virallista tyyliopasta. Tämän tarkoituksena on helpottaa päivittämistä tekemällä koodista enemmän Angularia muistuttavampaa. Tämä helpottaa myös kehittäjien työtä, sillä tyylioppaan mukaisesti kirjoitettua AngularJS-komponenttia voidaan myöhemmin käyttää upgradeModulen avulla suoraan myös uudemman Angularin kanssa. Esimerkkisovellus noudattaa virallisen tyylioppaan tyyliä. (Upgrading from AngularJS to Angular n.d.)

TypeScript

Angular on kirjoitettu TypeScript-ohjelmointikielellä, joten JavaScript kannattaakin jo heti päivityksen alkuvaiheessa vaihtaa käyttämään TypeScriptiä. Tämä asennus ei edellytä lainkaan koodimuutoksia, joten tämä vaihe on melko yksinkertainen. Jotta AngularJS-sovellukseen saadaan otettua mukaan TypeScriptin tuomat hyödyt, tarvitsee se ensin asentaa ja ottaa käyttöön. Tämän asennuksen voi tehdä esimerkiksi npm-pakettienhallintaa hyödyntäen ajamalla komentorivillä ”npm install typescript --save-dev”-komennon. TypeScript tarvitsee toimiakseen json-muodossa olevan konfiguraatiotiedoston nimeltään tsconfig.json. Tämän konfiguraatiotiedoston perusteella

käännös JavaScriptiksi toteutetaan. Alla olevassa esimerkissä näkyy esimerkki konfiguraatiotiedostosta, jonka perusteella TypeScript käännetään lopuksi JavaScriptin ES5-versioon (ks. kuvio 2). (TypeScript Configuration n.d.)

```
1  {
2    "compilerOptions": {
3      "target": "es5",
4      "module": "commonjs",
5      "moduleResolution": "node",
6      "sourceMap": true,
7      "emitDecoratorMetadata": true,
8      "experimentalDecorators": true,
9      "lib": [
10       "es2015",
11       "dom"
12     ],
13     "noImplicitAny": true,
14     "suppressImplicitAnyIndexErrors": true
15   }
16 }
```

Kuvio 2. Esimerkki tsconfig.json-tiedostosta

Kun TypeScript on saatu asennettua onnistuneesti ja konfiguraatiotiedosto on lisätty mukaan, voidaan testata kääntäjän toiminta suorittamalla komento "tsc". Komento ei toimi, jos yhtään .ts-tiedostopäätteellä löytyvää tiedostoa ei löydy projektirakenteesta. Tämän takia voidaankin aloittaa tiedostopäätteiden nimeäminen uudelleen .js-päätteestä .ts-päätteeksi. Kun tiedostopäätteet ovat muotoa .ts, voidaan komennolla "tsc" kääntää kyseiset tiedostot JavaScript-muotoon, jolloin projektirakenteessa näkyy .ts-tiedostojen lisäksi myös käännetyt .js-tiedostot. (Upgrading from AngularJS to Angular n.d.)

Onnistuneen TypeScriptin asennuksen ja tiedostopäätteiden muuttamisen jälkeen konsoliin tulee aluksi varoituksia, sillä nykyisessä koodipohjassa ei olla määriteltä muuttujien tyyppejä. Tästä eteenpäin onkin hyvä käytäntö aina muistaa ilmoittaa

muuttujien sekä funktioiden tyyppi. Tämä onnistuu määrittämällä muuttujan esimerkiksi `"title: string;"`, jossa kaksoispisteen jälkeinen osa määrittää kyseessä olevan string-tyyppinen muuttuja. TypeScript mahdollistaa myös luokka rakenteen käyttämisen eri komponenteille, joka ei ole ES5-JavaScriptissä muuten mahdollista. (Upgrading from AngularJS to Angular n.d.)

5.2 Angularin asennus

Seuraavaksi vuorossa on uudemman Angularin asennus sovellukseen. Tässä vaiheessa on hyvä tapa ladata GitHubista Angular quickstart-projektin tiedostot omaan kehitysympäristöön. Tämä projekti sisältää kirjoitushetkellä hyvän rakenteen uudelle Angular version 4.3.4-sovellukselle. Projektin mukana on package.json-tiedosto, johon määritellään riippuvuudet paketteihin ja mukana on myös määrittelyt `"@angular"`-alkuisiin paketteihin. Nämä tulisi kopioida myös päivitettävän AngularJS-sovelluksen package.json-tiedostoon. Tärkeä lisäys on tässä tapauksessa myös `"@angular/upgrade": "~4.3.4"`, sillä tämä paketti mahdollistaa hybridisovelluksen ajamisen. (Upgrading from AngularJS to Angular n.d.)

Tärkeä osa on myös moduulien lataaja, tässä esimerkissä käytetään SystemJS. Angular-moduulit ladataan tämän avulla, jolloin sovelluksessa voivat toimia samanaikaisesti sekä vanhempi AngularJS sekä uudempi Angular. SystemJS mahdollistaa myös tiedostojen laiskan lataamisen, eli tiedostojen lataaminen käyttöön vasta kun niitä tarvitaan. Tämä lyhentää sovelluksen ensimmäistä käynnistys aikaa, kun kaikkia tiedostoja ei ladata kerralla. Tähän tarkoitukseen on tarjolla myös muita moduulien lataajia laajemmilla toiminnallisuuksilla, esimerkiksi Webpack. Webpack vaatii paljon enemmän opettelua kuin SystemJS. SystemJS tulee quickstart-projektin mukana, joten tässä päivityksessä hyödynnetään tässä kohdassa SystemJS. Tämä vaatii toimiakseen myös systemjs.config.js-tiedoston, johon määritellään mistä kansioista sovelluksen tiedostot löytyvät ja mistä Angular-paketit haetaan sovelluksen käytettäviksi. Tämän konfiguraatitiedoston mukana tulisi olla samat pakettien määrittelyt kuin package.json-tiedostosta löytyy `"@angular"`-alkuisena. Angularin kannalta erityisen tärkeä tiedosto on hyvien käytänteiden mukaisesti yleensä nimetty `"main.ts"`, joka

hoitaa varsinaisen sovelluksen kääntämisen Angular-sovellukseksi ja mahdollistaa sovelluksen käynnistymisen. Tämä kyseinen skripti on rakenteeltaan hyvin yksinkertainen (ks. kuvio 3). (Upgrading from AngularJS to Angular n.d.)

```
1 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2
3 import { AppModule } from './app.module';
4
5 platformBrowserDynamic().bootstrapModule(AppModule);
6
```

Kuvio 3. Esimerkki Angularin main.ts-tiedostosta

Hybridisovelluksen ajaminen

Kun uudet Angular-paketit ja moduulien lataaja on saatu onnistuneesti asennettua, voidaan aloittaa hybridisovelluksen käyttöönotto. Tässä vaiheessa on hyvä tapa nimetä vanha AngularJS-pohjainen "app.module"-tiedosto "app.module.ajs"-tiedostoksi, jotta nimestä voi heti päätellä kumman moduuli on kyseessä. Tässä vaiheessa luodaan myös uusi "app.module"-tiedosto, joka toimii uudemman Angularin tyylillä. Tässä uudessa tiedostossa ylikirjoitetaan Angularin oletus bootstrap-metodin kutsu ja korvataan se upgrade-moduulin bootstrap-metodin kutsulla. Tämän jälkeen Angular-sovellus pystyy käsittelemään myös AngularJS-tyylisiä komponentteja ja palveluita, jotka käyttävät upgrade-moduulia. (Upgrading from AngularJS to Angular n.d.)

```

1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { UpgradeModule } from '@angular/upgrade/static';
4
5  @NgModule({
6    imports: [
7      BrowserModule,
8      UpgradeModule
9    ],
10  })
11  export class AppModule {
12    constructor(private upgrade: UpgradeModule) { }
13    ngDoBootstrap() {
14      this.upgrade.bootstrap(document.documentElement, ['app']);
15    }
16  }
17

```

Kuvio 4. Sovelluksen alustus hybridisovelluksena

5.3 Päivitä palvelut

Angularissa palvelut luodaan käyttäen selitettä `@Injectable`-luokan kanssa, jonka avulla määritetään kyseessä olevan palvelu, josta haetaan dataa esimerkiksi http-kutsun avulla. Kun palvelun muoto on muutettu käyttämään `@Injectable`a ja uudemman Angularin muotoa, tulee palvelu rekisteröidä ”angular.factory”-metodilla ja `downgradeInjectable`n avulla toimimaan AngularJS-komponenttien kanssa yhteensopivana. Tämän jälkeen on tärkeää muistaa poistaa index.html-tiedostosta skriptin määrittely ”<script src=’core/phone/phone.service.js’></script>”, sillä muutoksen myötä sitä ei tarvitse enää määritellä. (Upgrading from AngularJS to Angular n.d.)

5.4 Päivitä komponentit

Kun palvelut on saatu päivitettyä, on vuorossa komponenttien päivitys. Ennen päivitystä palvelut ovat Angular-muotoisia ja komponenttien tulisi olla näiden palvelujen kanssa yhteensopivia toimiakseen. Tässä vaiheessa komponentti sisältää PhoneListController-nimisen käsittelijän ja staattisen `$inject`-nimisen muuttujan, joka injekttoi palvelun (ks. kuvio 5). (Upgrading from AngularJS to Angular n.d.)

```

1  declare var angular: angular.IAngularStatic;
2  import { Phone, PhoneData } from '../core/phone/phone.service';
3
4  class PhoneListController {
5      phones: PhoneData[];
6      orderProp: string;
7
8      static $inject = ['phone'];
9      constructor(phone: Phone) {
10         phone.query().subscribe(phones => {
11             this.phones = phones;
12         });
13         this.orderProp = 'age';
14     }
15 }
16
17 angular.
18     module('phoneList').
19     component('phoneList', {
20         templateUrl: './phone-list/phone-list.template.html',
21         controller: PhoneListController
22     });
23

```

Kuvio 5. Esimerkkisovelluksen AngularJS-komponentti

Komponenttien päivitys tehdään yksi kerrallaan ja tässä vaiheessa komponentit alennetaan AngularJS-yhteensopivaksi upgrade-moduulin avulla (ks. kuvio 6). Komponenttien alennukset poistetaan vasta kun kaikki muut komponentit ovat päivitetty ja AngularJS-sovelluskehystä ei enää tarvita. Staattista \$inject-muuttujaa ei enää tarvitse määritellä, joten se voidaan poistaa. Angularissa komponentit määritellään @Component-ohjeistuksella, joten käsittelijä luokka voidaan korvata tällä uudella tyyllillä (ks. kuvio 6). Kun komponentti on saatu onnistuneesti päivitettyä, voidaan vanha skriptimäärittely poistaa index.html-tiedostosta. Tämän lisäksi lisätään Angularin app-moduuliin määrittely uudelle komponentille, sillä tämä uusi komponentti on osa Angular-sovellusta. Alennetut komponentit tulee määritellä myös moduulin entryComponents-nimisessä taulukossa. (Upgrading from AngularJS to Angular n.d.)

```

1  declare var angular: angular.IAngularStatic;
2  import { downgradeComponent } from '@angular/upgrade/static';
3
4  import { Phone, PhoneData } from '../core/phone/phone.service';
5  import { Component } from '@angular/core';
6
7  @Component({
8      selector: 'phone-list',
9      templateUrl: './phone-list/phone-list.template.html'
10 })
11 export class PhoneListComponent {
12     phones: PhoneData[];
13     query: string;
14     orderProp: string;
15
16     constructor(phone: Phone) {
17         phone.query().subscribe(phones => {
18             this.phones = phones;
19         });
20         this.orderProp = 'age';
21     }
22 }
23
24 angular.module('phoneList')
25     .directive(
26         'phoneList',
27         downgradeComponent({component: PhoneListComponent}) as angular.IDirectiveFactory
28     );
29

```

Kuvio 6. Esimerkkisovelluksen päivitetty AngularJS-yhteensopiva Angular-komponentti

Komponenttien päivitykseen olennaisena osana kuuluu myös komponenttiin liitetyn HTML-mallipohjan muuttaminen Angular-muotoon. Tässä vaiheessa on tärkeää tiedostaa, että kaikki AngularJS-direktiivit eivät ole enää käytettävissä uudemmissa versioissa, joten näiden toiminnallisuudet pitää määritellä TypeScript-koodissa. Tämä muutos johtuu näiden putkien suorituskyvyn tarpeesta, sillä näitä kutsutaan usein monia kertoja päällekkäin ja suuremmissa taulukoissa tämä on raskasta (No FilterPipe or OrderByPipe n.d). Direktiivien syntaksi on myös muuttunut matkan varrella. Esimerkiksi kuvan lähdettä ei määritellä enää ng-src-direktiivillä, vaan se on korvattu [src]-direktiivillä. Toinen paljon käytetty direktiivi ng-repeat on myös korvattu uudemmalla syntaksilla. Kun halutaan käydä taulukko läpi, Angularissa käytetään *ngFor-määrittelyä (ks. kuvio 7). (Upgrading from AngularJS to Angular n.d.)

```

<ul class="phones">

  <!-- AngularJS -->
  <!-- <li ng-repeat="phone in $ctrl.phones | filter:$ctrl.query | orderBy:$ctrl.orderProp"
    class="thumbnail phone-list-item"> -->

  <!-- Angular -->
  <li *ngFor="let phone of getPhones()" class="thumbnail phone-list-item">

    <a href="/#!/phones/{{phone.id}}" class="thumb">

      <!-- AngularJS -->
      <!--  -->

      <!-- Angular -->
      <img [src]="phone.imageUrl" [alt]="phone.name" />

    </a>
    <a href="/#!/phones/{{phone.id}}" class="name">{{phone.name}}</a>
    <p>{{phone.snippet}}</p>
  </li>
</ul>

```

Kuvio 7. Esimerkkisovelluksen päivitetyn komponentin HTML-mallipohja

Angular reititin

Vanha reititys ei toimi uuden Angularin kanssa, joten käyttöön pitää ottaa uuden-
tyyppinen reititin. Uutta reititystä varten on suositeltavaa tehdä juurikomponentti ja
nimetä se hyvien käytänteiden mukaisesti AppComponentiksi. Tähän juurikomp-
ponenttiin liitetään valmis RouterOutlet komponentti, jonka tarkoituksena on toimia
reitityksen näkymän sijoituspaikkana. Tämän lisäksi index.html-tiedostoon lisätään
tämä uusi juurikomponentti ja sen mukana tuleva reititetty näkymä. Näillä korvataan
vanha ng-view AngularJS-direktiivi. Reititin tulee vielä konfiguroida ja määrittää
mitkä komponentit näytetään aktiivisella reitityksellä. Tätä varten suositellaan luota-
vaksi reititysmoduuli nimeltään AppRoutingModuleModule (ks. kuvio 8). Tämä moduuli täy-
tyy vielä muiden moduulien tapaan määritellä juurimoduulin imports-kohdassa ja de-
clarations-kohtaan täytyy lisätä myös juurikomponentti. Kun tämä on tehty, voidaan
linkkeihin sitoa [routerLink]-arvo, jonka avulla kerrotaan reitittimelle aktivoitu osoite.
Jos komponentissa käytetään esimerkiksi id arvoa, joka tarvitsee hakea osoiteken-
tstä, voidaan se hakea injektoimalla ActivatedRoute-objekti reitittimeltä. Tätä voi-
daan käyttää esimerkiksi "phone.get(activatedRoute.snapshot.paramMap.get('phoneId'))". (Upgrading from AngularJS to Angular n.d.)

```

1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { APP_BASE_HREF, HashLocationStrategy, LocationStrategy } from '@angular/common';
4
5  import { PhoneDetailComponent } from './phone-detail/phone-detail.component';
6  import { PhoneListComponent } from './phone-list/phone-list.component';
7
8  const routes: Routes = [
9    { path: '', redirectTo: 'phones', pathMatch: 'full' },
10   { path: 'phones', component: PhoneListComponent },
11   { path: 'phones/:phoneId', component: PhoneDetailComponent }
12 ];
13
14 @NgModule({
15   imports: [ RouterModule.forRoot(routes) ],
16   exports: [ RouterModule ],
17   providers: [
18     { provide: APP_BASE_HREF, useValue: '/' },
19     { provide: LocationStrategy, useClass: HashLocationStrategy },
20   ]
21 })
22 export class AppRoutingModule { }
23

```

Kuvio 8. Esimerkki Angular-reititysmoduulista

5.5 AngularJS poistaminen

Kun koko sovellus on saatu päivitettyä Angular-sovellukseksi, voidaan kaikki AngularJS-viittaukset poistaa sovelluksesta. Tämä kannattaa aloittaa Angularin juurimoduulissa tapahtuvasta upgrade-moduulin bootstrap-metodin kutsusta, sillä sovellusta ei tarvitse enää ajaa hybridisovelluksena. Seuraavaksi on hyvä poistaa kaikki viittaukset upgrade-moduulin kutsuihin ja poistaa koko moduuli ja AngularJS-tyypitykset seuraavilla komennoilla:

- "npm uninstall @angular/upgrade --save"
- "npm uninstall @types/angular @types/angular-animate @types/angular-cookies @types/angular-mocks @types/angular-resource @types/angular-route @types/angular-sanitize --save-dev".

Tämän lisäksi voidaan poistaa index.html-tiedostosta tarpeettomat viittaukset esimerkiksi vanhoihin AngularJS-komponentteihin, sillä näitä ei enää tarvita ja nämä viittaukset saattavat unohtua poistaa. Sovellus on nyt täysin Angular-sovellus ja sitä voidaan lähteä kehittämään Angularin tyylillä. (Upgrading from AngularJS to Angular n.d.)

6 Tutkimustulokset

6.1 Tutkimuskohteen esittely

Tutkimuskohde on AngularJS-sovelluskehystä hyödyntäen rakennettu JavaScript-sovellus. Sovellus on ollut useamman henkilön tiimin kehityksessä kirjoitushetkellä noin viiden vuoden ajan. Koodipohja on kasvanut ajan myötä tehden sovelluskehysten päivittämisestä haastavampaa.

AngularJS

Tutkimuskohteessa on kirjoitushetkellä käytössä AngularJS-versio 1.3.13. Tässä versiossa ei ole vielä mahdollista käyttää komponenttipohjaista toteutusta käsittelijöille, joten sovellus on toteutettu käyttäen controller-tyyppisiä käsittelijöitä käyttäen \$scopea. AngularJS:n mukana tuleva reititin on korvattu kolmannen osapuolen ui-router-reitittimellä. Tämän reitittimen avulla voidaan tehdä täysin samoja asioita kuin mukana tulevalla reitittimellä, mutta sen avulla pystytään myös laajentamaan reitittimen toimintaa. Tämä mahdollistaa oletusreitittimeen verrattuna esimerkiksi tilan välityksen muille näkymille ja mahdollisuuden luoda sisäkkäisiä näkymiä (engl. nested view). Sovelluksen logiikka on jaettu moniin pieniin moduuleihin, mutta näiden moduulien koodit ovat jaettu tiedostoihin tyyppin mukaan. Tämä tarkoittaa, että samassa tiedostossa on useita käsittelijöitä tai palveluita.

Sovelluksen tyylit ovat SASS-muotoisia, eli tyylitiedostot ovat .scss-päätteisiä. Sass on CSS3-tyylin ylijoukko, eli sitä voidaan kirjoittaa myös puhtaasti CSS-tyylisesti. Sass mahdollistaa esimerkiksi muuttujien määrittelyn tyyleille. Lopuksi Sass käännetään tavalliseksi CSS3:ksi, jotta se toimii selaimissa. Tutkimuskohteessa tähän käytetään gulp-sass lisäosaa.

Pakettienhallinta

Tutkimuskohteessa on kirjoitushetkellä käytössä bower ja npm pakettienhallinnassa. Npm on todella laajasti käytetty pakettienhallinnan työkalu, jonka avulla voidaan

asentaa kolmannen osapuolen avoimen lähdekoodin JavaScript-paketteja sovellukseen (What is npm? 2011). Npm-pakettienhallinnassa käytössä olevat paketit määritellään package.json-nimiseen tiedostoon. Nämä paketit voidaan asentaa ajamalla komento ”npm install”. Yksittäisiä paketteja voidaan asentaa asettamalla asennuskomentoon parametriksi paketin nimi.

Tutkimuskohteessa kirjoitushetkellä käytössä oleva Bower on npm kaltainen pakettienhallinta työkalu, joka on vanhentunut. Bowerin kehittäjäjoukko suosittelee käyttämään jotain muuta pakettienhallintaa uusille projekteille (Stankiewicz 2017). Bower on pakettienhallinnan työkalu, jota on käytetty vanhoissa AngularJS-sovelluksissa pakettienhallinnassa. Tutkimuskohteessa AngularJS ja siihen liittyvät paketit ovat asennettu bowerin avulla.

Gulp

AngularJS-sovelluskehyksessä ei ole oletuksena automaattista muutosten tarkkailua, joten tähän on tarjolla kolmannen osapuolen tekemiä sovelluksia automatisoimaan build-prosessia koodimuutoksen mukaan. Tutkimuskohteena olevassa sovelluksessa on apuna Gulp. Modernien web-sovelluksien kehittämisessä on monia usein toistuvia tehtäviä, kuten koodin minifiontia ja paikallisen testaus palvelimen ajamista. Gulp tarjoaa apua näihin tehtäviin automatisoimalla niitä tarpeen mukaan omalla konfiguraatiolla. Esimerkiksi paikallisesti kehittäessä Gulp voi tarkkailla koodimuutoksia ja päivittää näkymän selaimessa muutoksen havaitessa. Gulp-tehtävät konfiguroidaan aina gulpfile.js-nimiseen konfiguraatiotiedostoon, joka on kirjoitettu JavaScriptillä. Gulpin avulla voidaan määrittää oletustehtäviä käyttämällä default-tehtävää, joka suoritetaan jokaisella kerralla, kun gulp-komentoa kutsutaan. Gulpissa on myös mahdollista ottaa käyttöön sen toimintaa laajentavia paketteja, jotka ovat nimetty gulp-etuliitteellä. (Neale 2016.)

6.2 Tutkimuskohteen päivitys

Angularin kehittäjäjoukko on käyttänyt paljon aikaa päivitysohjeiden ja erilaisten helpottavien työkalujen tekemiseen. Päivityksen avuksi on tehty kehittäjäjoukon toimesta komentorivityökalu, jonka avulla voidaan analysoida sovelluksen rakenne.

Tämä työkalu antaa suosituksen, miten sovellusta kannattaa lähteä päivittämään ja mitä ennen päivittämisen aloitusta kannattaa muuttaa. Työkalu voidaan asentaa globaalisti ajamalla komento ”npm install -g ngma”. Tämän jälkeen voidaan suorittaa komento ”ngma” AngularJS-sovelluksen sijainnissa. Tämän lisäksi päivityksen avuksi on luotu foorumi, jossa voidaan kysyä yhteisön apua päivitykseen liittyen. (Olson 2018.)

6.2.1 Valmistelu

Koodipohja tyylioppaan mukaiseksi

Kaikkein tärkein osa tulevan päivityksen kannalta on tyylioppaan ensimmäinen kohta, eli yhden sääntö. Tämä tarkoittaa käytännössä sitä, että jokainen AngularJS-sovelluksen osa on omassa JavaScript-tiedostossaan. Tiedostot tulisi nimetä niin että heti nimestä huomaa mikä osa on kyseessä. Suositeltua on käyttää tiedostomuotoa ”toiminto.tyyppi.js” (Papa n.d.) Esimerkiksi sisäänkirjautumisen käsittelijän tulisi sijaita ”login.controller.js”-nimisen tiedoston sisällä. Virallinen tyyliopas sisältää myös paljon muita hyviä käytänteitä päivittämiseen liittyen. Tyylioppaan tarkoitus on antaa suuntaa AngularJS-sovelluksen tyyliä, jotta sovelluksen rakenne muistuttaisi enemmän uudemman Angularin rakennetta. Tämä taas helpottaa päivityksen toteutusta.

Bower

Boweria on käytetty vanhojen projektien pakettienhallinnassa yhdessä npm kanssa. Osa sovelluksen päivitystä onkin bower-pakettien korvaaminen vastaavilla npm-paketeilla. Tässä kohdassa on tärkeää huomioida, että bowerin muoto nimetä paketteja on erilainen kuin npm ja pakettien versionumerot saattavat myös erota toisistaan. Npm-pakettien nimeäminen on aina pienellä kirjoitettuna ja sanat on erotettu toisistaan väliviivalla, eli esimerkiksi ”angular-ui-router”. Bowerissa nimeäminen ei aina noudata tätä tyyliä, vaan joukossa voi myös olla isoja kirjaimia. Tämän lisäksi bowerin versionumeroinnissa voi olla mukana paketin nimi, esimerkiksi ”ui-router#^0.4.2”. Bowerin pakettien dependencies-listaus voidaan kopioida package.json-tiedostoon, johon pitää muokata npm-muotoiseksi nimi ja versionumero. Pakettien asennusvaiheessa ”npm install”-komennolla konsoliin tulee virhe, jos paketin muoto on virheelinen tai pakettia ei löydy.

Webpack moduulien lataaja

Kun koodipohja on saatu päivitettyä vastaamaan tyyliopasta, seuraavaksi vuorossa on TypeScriptin käyttöönotto. TypeScript mahdollistaa JavaScriptin uusien ominaisuuksien käyttöönoton ES5-yhteensopivana. Webpack on työkalu, joka mahdollistaa moduulien lataamisen ja sitä tai muuta vastaavaa suositellaankin käytettävän moduulien lataamisessa ja TypeScriptin kääntämisessä. Webpack ja TypeScript voidaan asentaa hyödyntäen komentoa `"npm install rimraf ts-loader typescript webpack --save-dev"`. Tässä kohdassa on hyvä asentaa myös rimraf-paketti, sillä sen avulla voidaan puhdistaa Webpackin luoma bundle.js-tiedosto. Eli Webpack luo bundle.js tiedoston aina uudelleen puhtaalta pöydältä. Webpack vaatii toimiakseen yhdessä TypeScriptin kanssa erillisen ts-loader paketin, jonka avulla TypeScript-moduulit voidaan ladata.

Kun paketit on saatu asennettua, tarvitaan TypeScriptille ja Webpackille konfiguraatiotiedostot. Konfiguraatiotiedosto TypeScriptille suositellaan nimettävän tsconfig.json-tiedostoksi. Tähän tiedostoon määritellään miten ja mihin .ts-tiedostot käännetään. Webpackin konfiguraatiotiedosto on muotoa webpack.config.js. Tähän konfiguraatiotiedostoon määritellään esimerkiksi mistä moduulit haetaan ja mihin rakennettu JavaScript-tiedosto sijoitetaan (ks. kuvio 9). Luotavan tiedoston nimeä voidaan myös muuttaa tässä kohdassa, mutta bundle.js on todella laajasti käytetty. Konfiguraatiotiedoston lisäksi tulee luoda tiedosto, jossa määritellään kaikki ladattavat moduulit. Tätä tiedostoa kutsutaan yleensä nimellä `"main.ts"`.

```
1  module.exports = {  
2    entry: "./src/app/main.ts",  
3    output: {  
4      filename: "src/dist/bundle.js"  
5    },  
6    resolve: {  
7      extensions: [".webpack.js", ".web.js", ".ts", ".tsx", ".js"]  
8    },  
9    module: {  
10     rules: [  
11       { test: /\.tsx?$/, loader: "ts-loader" }  
12     ]  
13   }  
14 };  
15
```

Kuvio 9. Webpackin konfigurointi

AngularJS-version päivitys

Angularin 2 version julkaisun jälkeen myös AngularJS oli vielä aktiivisen kehityksen alla. Angularin hyviä käytänteitä alettiin ottaa käyttöön myös AngularJS-puolella, joten päivittämällä AngularJS viimeisimpään versioon tuo sovelluskehystä lähemmäs Angularia. Merkittävin hyöty päivityksen kannalta on 1.5.x-versiossa mukana tullut komponenttirajapinta, jonka avulla voidaan luoda komponentteja. Uudemman version asennus tapahtuu käytännössä päivittämällä pakettienhallintaan pakettien versionumero ja ajamalla asennuskomennon. Päivittäessä uudempaan versioon on tärkeää ottaa huomioon, että osa koodista saattaa olla vanhentunutta, eikä toimi ilman koodimuutoksia. Näihin muutoksiin korjaukset löytyvät virallisesta dokumentaatiosta ”breaking changes”-kappaleesta kunkin 1.x-version kohdalla.

AngularJS-komponenttirakenne

AngularJS-versiossa 1.5.x esiteltiin uusi komponenttirakenne, jonka tarkoituksena on korvata sekä direktiivit mallipohjien kanssa että käsittelijät HTML ng-controller-direktiivin kanssa. Komponenttirakenteessa käytetäänkin vielä käsittelijää, mutta se on vain yksi osa kutakin komponenttia. Komponenttirakenne käyttää oletuksena controllerAs-syntaksia, jolloin \$scope-objektia ei enää tarvita. Tämän objektin muuttujat voidaan korvata this-avainsanalla. Hyvä käytäntö tässä on, että ei määritä ja käytä muuttujia suoraan this-avainsanan avulla. Tämän sijaan viittaus tulisi määritellä heti alussa esimerkiksi vm-nimiseen muuttujaan, jolloin this ei viittaisi vahingossakaan väärään paikkaan. Tämä komponenttirakenne mahdollistaa myöhemmin upgrade-moduulin käyttämisen, jonka avulla voidaan alentaa uudemman Angularin komponentit yhteensopiviksi AngularJS:n kanssa. Komponentteja voidaan käyttää samaan tyyliin kuin direktiivejä eli mallipohjaan tai reitittimeen määritellään esimerkiksi ”<my-component></my-component>”.

Modernisointi

Kun TypeScript on otettu käyttöön, voidaan koodissa ottaa käyttöön moderneja tapoja kirjoittaa koodia, joita JavaScriptin ES5-versio ei vielä tue. Uudempi Angular on kirjoitettu TypeScriptillä ja siinä hyödynnetään luokkarakennetta komponenteille.

Vanhaan koodipohjaan voidaan myös ottaa käyttöön luokat, sillä tämä helpottaa uusien Angular-komponenttien kirjoittamista myöhemmin. Tämä tarkoittaa komponentin käsittelijän vaihtamista funktiosta luokaksi. Tämän lisäksi tyyppitys helpottaa mahdollisten virheiden huomaamista jo kehitysvaiheessa. AngularJS sisältää ngResource-moduulin sisäänrakennettuna, mutta tälle ei ole suoraa vastinetta uudemmassa Angularissa. Tätä moduulia käytetään REST-rajapinnan kautta tekemään http-kutsuja, joiden avulla voidaan hakea, lisätä, päivittää tai poistaa palvelimelta tietoja (\$resource n.d.). Tämä ngResource-moduuli kannattaakin korvata \$http-palvelulla, joka mahdollistaa samat kutsut kuin ngResource-moduuli. Uudemmassa Angularissa on vastaavanlainen http-palvelu sisäänrakennettuna ja sen käyttöönotto on helppoa, jos jo AngularJS-vaiheessa on otettu se käyttöön. Palvelu \$q on palvelu, jota käytetään asynkronisten funktioiden kutsumiseen. Tähän soveltuu myös ES6-JavaScriptin natiivi lupaus, joita on mahdollista käyttää TypeScriptin ansiosta.

6.2.2 Angularin asennus

Kun AngularJS on saatu päivitettyä komponenttirakenteiseksi ja uudempaan versioon, voidaan aloittaa Angularin käyttöönotto. Angular voidaan asentaa määrittelemällä tarvittavat paketit package.json-tiedostoon ja ajamalla komento "npm install" (ks. kuvio 10). Pakettilistauksen ja hyvän rakenteen esimerkin uudelle Angular-sovellukselle voi myös hakea luomalla uuden projektin Angular CLI:n avulla. Tämän lisäksi tarvitsee määritellä polyfills.ts-nimiseen tiedostoon uudemman JavaScriptin ominaisuudet, joiden oletetaan toimivan natiivisti uusimmilla selaimilla. Tämä varmistaa sovelluksen tuen myös vanhemmille selaimille. Polyfills-tiedosto tarvitsee importoida main.ts-tiedostossa sovelluksen käyttöö.

```

"dependencies": {
  "@angular/animations": "^8.0.0",
  "@angular/common": "^8.0.0",
  "@angular/compiler": "^8.0.0",
  "@angular/compiler-cli": "^8.0.0",
  "@angular/core": "^8.0.0",
  "@angular/forms": "^8.0.0",
  "@angular/platform-browser": "^8.0.0",
  "@angular/platform-browser-dynamic": "^8.0.0",
  "@angular/platform-server": "^8.0.0",
  "@angular/router": "^8.0.0",
  "@angular/upgrade": "^8.0.0",
  "angular-in-memory-web-api": "~0.8.0",
  "core-js": "^3.1.14",
  "rxjs": "^6.5.2",
  "web-animations-js": "^2.3.2",
  "zone.js": "^0.10.1",

```

Kuvio 10. Angular asennukseen liittyvät paketit

AngularJS-sovellukset käyttävät ng-app-direktiiviä sovelluksen näyttämiseen. Tämän direktiivin taustalla on AngularJS:n bootstrap-metodi. Angular käyttää myös bootstrap-kutsua, mutta se tapahtuu hieman eri tavalla. Tässä vaiheessa ng-app-direktiivi voidaan poistaa index.html-tiedostosta. Ng-app direktiivin tilalle luodaan juurimoduuli nimeltään AppModule. Tässä vaiheessa Angularin oletus bootstrap-metodi ylikirjoitetaan ja käytetään upgrade-moduulin bootstrap-metodia. Tämän kutsun avulla myös vanhemmat AngularJS osat ovat yhteensopivia alennettujen Angularin kanssa (ks. kuvio 11). Tämä juurimoduuli haetaan myös main.ts-tiedoston käyttöön importin avulla.

```

51 export class AppModule {
52   constructor(private upgrade: UpgradeModule) { }
53   // Override Angular bootstrap and use upgrade module to bootstrap
54   ngDoBootstrap() {
55     console.log("Bootstrapping in Hybrid mode with Angular & AngularJS");
56     this.upgrade.bootstrap(document.body, ['app']);
57   }
58 }
59

```

Kuvio 11. Oletus bootstrap-metodin korvaus hybridiversiolla

6.2.3 Palveluiden päivitys

Aikaisemmin ngResourcen korvaaminen AngularJS:n \$http-palvelulla mahdollistaa helpomman Angular-HttpClientin käyttöönoton. HttpClient on samantyylinen kuin AngularJS:n \$http. Uusimmissa versioissa HttpClient tulee Angularin common-paketin mukana. Se voidaan ottaa käyttöön käyttäen `import {HttpClient} from "@angular/common/http";`. Normaalisti Angularin palvelut määritellään käyttäen @Injectable-määrittelyä luokalle, mutta hybridisovelluksen määrittämisessä kannattaa käyttää konstruktorin muuttujien määrittelyssä @Inject-määrittelyä. HttpClientin tapauksessa se otetaan konstruktorissa käyttöön `constructor(@Inject(HttpClient) private http: HttpClient) { }`.

Rakenteeltaan AngularJS:n \$http ja Angularin HttpClient ovat samankaltaiset. Huomioitavana tässä kuitenkin on, että HttpClient palauttaa tarkkailijan, kun taas \$http palauttaa lupauksen. Tässä vaiheessa voidaan halutessa muokata komponenttien logiikka käyttämään tarkkailijoita. Jos palvelut halutaan suoraan toimiviksi olemassa olevien AngularJS-komponenttien kanssa, voidaan tähän kuitenkin käyttää rxjs-operaattoria toPromise(). Tämä tarvitsee hakea palvelun käyttöön käyttäen `import 'rxjs/add/operator/toPromise';`.

Jotta Angular-muotoista palvelua voidaan käyttää yhdessä AngularJS:n kanssa, tulee palvelu alentaa downgradeInjectable-metodin avulla. AngularJS moduulille määritellään factory-tyyppinen palvelu, joka on alennettu Angular palvelu (ks. kuvio 12).

Myöhemmin, kun palvelua ei käytetä enää muissa kuin Angularin omissa palveluissa ja komponenteissa, downgradeInjectable-määrittelyä ei enää tarvita.

```
120  angular
121    .module("app")
122    .factory("ExampleService", downgradeInjectable(ExampleService));
123
```

Kuvio 12. Angular-palvelun alentaminen AngularJS-yhteensopivaksi

Kolmannen osapuolen moduulien, kuten toasterin kanssa voidaan joko etsiä vastaava Angularin versio, kirjoittaa se itse uudelleen tai väliaikaisesti ylentää se Angular-yhteensopivaksi. Nämä väliaikaiset ylennykset kannattaa tehdä erillisessä tiedostossa, joka on nimettynä esimerkiksi "ajs-upgraded-providers.ts". Ylennyksen ideana on määrittää moduuli yhteensopivaksi Angularin riippuvuusinjektion kanssa, samalla säilyttäen yhteensopivuuden myös AngularJS kanssa (ks. kuvio 13). Tämä moduuli määrittellään myös sen Angular-moduulin providers-kohdassa, jossa sitä käytetään. Paras tapa kolmannen osapuolen pakettien päivittämiseen on etsiä vastaava Angular-versio, sillä se toimii myös ilman vanhan AngularJS:n versiota myöhemmissä vaiheissa. Yleensä nämä paketit löytyvät ngx- tai angular2-etuliittellä. Nämä paketit toimivat yleensä kuitenkin myös uudempien versioiden kanssa.

```
1  import {InjectionToken} from '@angular/core';
2
3  export const Toaster = new InjectionToken("Toaster");
4
5  export function toasterServiceFactory(i: any){
6      return i.get('toaster');
7  }
8  export const toasterServiceProvider = {
9      provide: Toaster,
10     useFactory: toasterServiceFactory,
11     deps: ['$injector']
12 };
13
```

Kuvio 13. Ylennetty kolmannen osapuolen AngularJS-moduuli

6.2.4 Komponenttien päivitys

Seuraavaksi vuorossa on komponenttien päivitys yksi kerrallaan Angular-muotoisiksi. Komponenttien päivityksen yhteydessä tulee eteen myös vanhojen AngularJS-filtterien uudelleenkirjoitus, sillä osa niistä on poistettu. Uudet komponentit tulee määrittellä Angular-moduulin declarations-listaukseen. Jotta nämä komponentit toimivat AngularJS-yhteensopivina, tulee nämä komponentit alentaa käyttäen upgrade-moduulin downgradeComponent-metodia. Tämän lisäksi nämä alennetut komponentit tulee määrittellä myös moduulin entryComponents-listauksessa. Kun komponenttia

lähdetään uudelleenkirjoittamaan Angular-muodossa, tarvitsee komponentin metatiedot määrittellä `@Component`-määrittelyllä. Komponenttien metatietoihin määritellään tiedot, jolla komponentti voidaan hakea näkymään. Myös mallipohja ja mahdolliset komponenttikohtaiset tyylit määritellään metatietoihin.

Komponenttien päivityksessä työläin tehtävä on mallipohjan muutos uuteen Angular-muotoon. AngularJS käyttää usein kaksisuuntaista datasidontaa, kun Angularissa suositetaan yksisuuntaista. Tämä tarkoittaa muutosta lomakkeiden (engl. forms) käsittelyssä. Angularissa mukana tulee lomakkeiden käsittelyyn esimerkiksi `FormGroup`, jonka avulla voidaan määrittellä lomakkeen logiikka TypeScript-koodissa vanhan mallipohjan käsittelyn tilalla. `FormGroup`in käyttö mahdollistaa reaktiivisten `rxjs`-operaattorien käyttämisen. Eli `ng-model`in tilalla voidaan käyttää esimerkiksi tarkkailijapohjaista arvonmuutoksen kuuntelijaa. AngularJS käytetään paljon `ng`-alkuisia direktiivejä mallipohjan tietojen käsittelyssä. Angularissa monen kirjoitusasu on muuttunut, joten `ng`-alkuiset direktiivit tulee korvata uudemman Angularin tyylillä toteuttaa nämä. Tässä hyvä lähtökohta on, jos mallipohjassa direktiivin kirjoitustyyli on esimerkiksi `muotoa ng-model`, tulee se korvata toimiakseen uudemmassa Angularissa. Virallisesta dokumentaatiosta löytyy suoraan vastaavat mallipohjan direktiivit AngularJS ja Angular välillä (AngularJS to Angular Concepts: Quick Reference n.d.).

Kun komponentti ja sen mallipohja on päivitetty Angular-muotoon, tarvitsee se vielä alentaa yhteensopivaksi vanhempien komponenttien kanssa. Tässä vaiheessa tämä komponentti määritellään AngularJS-direktiiviksi. Direktiiville määritellään komponentti käyttäen `upgrade-moduulin` `downgradeComponent`-metodia. Kaikki nämä direktiivimääritellyt komponentit tulee vielä määrittellä moduulin `entryComponents`-listaukseen. Tämän lisäksi suurin osa filttäreistä tulee uudelleenkirjoittaa putkiksi, sillä vastaavia ei välttämättä ole uudemmassa Angularissa. Putket määritellään `@Pipe`-määrittelyllä ja metatietoihin määritellään nimi, jolla putki haetaan mallipohjassa. Uudet putket määritellään moduulin `declarations`-listaukseen komponenttien kanssa. Myös AngularJS-direktiivit, joilla on oma näkymä, tarvitsee päivittää komponenteiksi käyttämällä `@Component`-määrittelyä. Direktiivien kohdalla AngularJS käyttää `bindings`-objektia, jonka tarkoituksena on määrittää arvon tulevan parametrina joltakin muulta elementiltä. Nämä voidaan korvata muuttujien `@Input`-

ja @Output-määrittelyllä. Tässä kohdassa on tärkeää huomioida oikea syntaksi komponentin mallipohjan määrittelyssä. Jos kyseessä on alennettu Angular-komponentti, tulee input-syntaksin noudattaa Angularin tyyliä ja määrittellä arvo hakasulkeissa. Kaksiosaisen nimen omaavien muuttujien kohdalla alennetussa komponentissa on erityinen, syntaksiltaan yhdistelty kirjoitusasu. Esimerkiksi muuttuja isLoading tulisi alennetun komponentin templaattissa kirjoittaa muodossa [is-loading], sillä AngularJS ei osaa käsitellä Angular-tyyppistä [isLoading]-määrittelyä. Ja koska kyseessä on Angular-komponentti, tarvitsee muuttujan määrittelyssä olla hakasulkeet.

Reititys

Tutkimuskohteen reititys on kirjoitushetkellä käsitelty ui-router-nimisellä kolmannen osapuolen paketilla. Tämä on erittäin paljon käytetty ratkaisu AngularJS-sovelluksissa, sillä se mahdollistaa paljon enemmän kuin alkuperäinen reititin. Uudemman Angularin mukana tuleva reitittimeen on kuitenkin tullut paljon lisää aikaisempaan verrattuna, joten sitä käytetään todella paljon kolmannen osapuolen ratkaisujen sijaan. Jotta ui-router-reitittimen tilaa voidaan vaihtaa alennetuista Angular-komponenteista, tarvitsee se ylentää väliaikaisesti toimimaan Angular-yhteensopivana (ks. kuvio 14). Tähän voidaan myös käyttää hybridi ui-router pakettia versiosta 3 eteenpäin, sillä se tukee Angularin upgrade-moduulia.

```
1 import {InjectionToken} from '@angular/core';
2
3 export const UIRouterState = new InjectionToken("UIRouterState");
4
5 export function uiRouterStateServiceFactory(i: any){
6     return i.get('$state');
7 }
8
9 export const uiRouterStateServiceProvider = {
10     provide: UIRouterState,
11     useFactory: uiRouterStateServiceFactory,
12     deps: ['$injector']
13 };
```

Kuvio 14. Väliaikaisesti ylennetty ui-router-tilanhallinta

Kun varsinaista reititystä lähdetään päivittämään uudemman Angularin tyyliiseksi, tarvitsee index.html-tiedostoon määritellä tagi `<base href="/">` head-osion alkuun. Tämän avulla Angularin reititin osaa käsitellä sille määritellyt reitit oikein. Aluksi on hyvä luoda reititysmoduuli, johon sovelluksen eri reitit määritellään. Tätä varten suositellaan alustettavan `appRoutes`-niminen muuttuja, johon määritellään taulukko-muodossa reittiobjektit. Tämä taulukko käydään läpi moduulin `imports`-kohdassa `RouterModule.forRoot()`-määrittelyllä. Tämän lisäksi kannattaa luoda juurikomponentti, johon reitittimen vaihtuva näkymä sijoitetaan. Tämä tapahtuu lisäämällä `<router-outlet>`-komponentti mallipohjaan. Tämä vastaa ui-routerin `ui-view`-direktiiviä ja näitä voi olla useita eri nimisiä, esimerkiksi `<router-outlet name="header"></router-outlet>`. Direktiivi `ui-sref` kertoo tällä hetkellä reitittimelle mikä reitti tulisi aktivoida. Nämä korvataan `[routerLink]`-määrittelyllä. TypeScript puolen koodissa käytetty `$state.go`-metodi voidaan korvata reitittimen `navigate`-metodilla. Reititysmoduuli tulee ensin injektoida luokan käyttöön konstruktorissa. Toinen ui-routerin kanssa käytetty palvelu on `$stateParams`, jonka avulla voidaan hakea aktiivisen reitin parametrejä. Angular-reitittimessä tämä määritellään käyttäen reitittimen mukana tulevaa `ActivatedRoute`-luokkaa. Tämä toimii hieman eri tavalla, sillä aktiivisen reitin parametrit ovat tarkkailija muodossa. Tarkkailijan arvo voidaan hakea parametreista `params.subscribe`-metodilla.

Kun reitin ja kaikki reitit on saatu onnistuneesti päivitettyä, sovellus on täysin Angular-sovellus. Tämä tarkoittaa sitä, että komponenttien alennukset ja `upgrade`-moduulin viittaukset voidaan poistaa komponenteista sekä palveluista. Tässä vaiheessa voidaan myös poistaa väliaikaisesti käytössä olleet ylennetyt AngularJS-osat.

6.2.5 Jälkityöt

Kun sovellus on saatu päivitettyä, voidaan aloittaa AngularJS-pakettien ja niiden viittausten poistaminen. Tämä on erittäin tärkeä osa päivitystä, sillä webpack käsittelee myös nämä tiedostot, vaikka niitä ei käytettäisi ollenkaan. Tämän lisäksi koodista tulee luettavamaa, kun vanhat koodit eivät ole enää mukana. Kun ollaan poistamassa vanhoja viittauksia ja paketteja, kannattaa säännöllisesti tarkistaa sovelluksen toiminta. Tämä on siltä varalta, jos jokin onkin käytössä päivitettyssä sovelluksessa. Osa

viittauksista on jo poistettu aikaisemmissa vaiheissa, mutta todennäköisesti AngularJS-juurimoduuli on vielä olemassa ja se voidaan poistaa. Myös omat filtterit voidaan poistaa, sillä nämä ovat korvattu putkilla. Tässä kohdassa on hyvä huomioida, että viittaukset polyfills-tiedostoon tulisi säilyttää. Package.json-tiedoston AngularJS-pakettien määrittelyn voi tässä vaiheessa poistaa, jotta käytöstä poistettuja paketteja ei asennettaisi uudelleen "npm install"-komennon yhteydessä.

7 Johtopäätökset

Verkossa on paljon ohjeistusta tässä tutkimuksessa käsiteltyyn aiheeseen. Ajan myötä aineiston ja esimerkkien määrä tähän liittyen kasvaa. Angularin kehittäjäjoukon tuki AngularJS-sovelluksen päivitykselle uudempaan Angulariin on todella merkittävä. Tiimi on esimerkiksi julkaissut erilaisia työkaluja, joiden tarkoituksena on tehdä tästä prosessista helpompaa. Tämän lisäksi Angularin ollessa avoimen lähdekoodin sovelluskehys, myös yhteisö on ollut aktiivisesti mukana kehittämässä sitä ja sen työkaluja. Päivityksen suunnittelu on tärkein osa päivitystä ja on erittäin tärkeää ottaa huomioon myös päivityksellä saavutettavissa oleva hyöty. Jos sovellus ei ole enää aktiivisen kehityksen alla, on todennäköistä, että päivityksestä saatava hyöty on pienempi kuin siihen käytetty aika.

Mitä AngularJS-sovelluksen päivittämiseen Angular-sovellukseksi tarvitaan?

Jokaisen AngularJS-sovelluksen rakenne on erilainen. Tämän takia päivittämisen tueksi tarvitaan paljon pohjatietoa ja tutkimusta. Jos sovellus on toteutettu vanhemmalla AngularJS-versiolla, ei mukana ole niin tiukkoja sääntöjä kuin uusimmissa versioissa. Tämän lisäksi tietoa hyvistä kehityskäytänteistä ja sovelluskehyksestä ylipäänsä ei ole ollut tarjolla läheskään yhtä paljoa kuin nykyään. Päivityksen yhteydessä kannattaa myös miettiä uudelleen sovelluksen arkkitehtuuria ja sen mahdollisia kehityskohtia. AngularJS-sovelluksissa käytetään paljon kolmannen osapuolen paketteja. Päivittämisen aikana nämä paketit tulee ottaa huomioon, sillä vastaavia paketteja ei mahdollisesti ole tarjolla Angularin puolella. Jos vastaavaa pakettia ei löydy Angularille, täytyy sen osalta tehdä korvaavia ratkaisuja.

Sovelluskehityksen päivittämistyyliin on useita eri vaihtoehtoja, joista valitaan tapauskohtaisesti sopivin. Useimmiten suurissa ja pitkään kehityksessä olleissa sovelluksissa valitaan hybridivaihtoehto, jossa sovelluksen siirtymävaiheessa on samanaikaisesti mukana molemmat sovelluskehikset. Joissakin tapauksissa koko sovelluksen uudelleenkirjoitus alusta alkaen voi olla parempi vaihtoehto. Uudelleenkirjoitus on hyvä vaihtoehto erityisesti sovelluksen koon ollessa pieni. Tämä mahdollistaa sovelluksen hyvien osien säilyttämisen ja huonojen osien korvaamisen paremmilla ratkaisuilla. Tämä kuitenkin vaatii paljon työtä ja tähän käytetty aika on pois uusien ominaisuuksien kehittämisestä.

Kuinka suuri työpanos ja osaaminen päivittämiseen tarvitaan?

Sovelluksen koko vaikuttaa tarvittavaan osaamiseen ja työpanokseen huomattavasti. Päivityksessä erityisen suuri hyöty on uudemman Angularin osaamisesta, sillä päivityksen edetessä vanha koodi päivitetään siihen. AngularJS-ymmärrys on myös eduksi, jotta vanhaa pohjaa voidaan päivittää modernimmaksi. Päivitysvaiheessa tästä on myös todella paljon hyötyä, sillä vanhojen AngularJS-ratkaisujen ymmärrys helpottaa korvaavan vaihtoehdon kehittämistä Angularin puolella.

Hybridivaihtoehdossa päivitys jakautuu kahteen selkeään osioon. Ensimmäisessä vaiheessa vanha koodipohja valmistellaan uusimpaan versioon soveltaen parhaita kehityskäytänteitä, jotta hybridisovelluksen käyttöönotto on mahdollista. Tämä vaihe tekee koodin luettavuudesta paljon helpompaa ja vaatii paljon AngularJS-osaamista. Tässä vaiheessa käyttöön otetaan myös moduulienlataaja, joten osaaminen esimerkiksi webpackin konfiguroinnista ja sen käyttämistä lisäosista auttaa tässä kohdassa huomattavasti. Riippuen sovelluksen rakenteesta, tämä ensimmäinen vaihe voi olla todella yksinkertainen prosessi. Yleensä näin ei kuitenkaan ole ja tähän vaaditaan suuri työpanos. Hyvä lähtökohta tähän vaiheeseen on ajatus, että sovelluksen tulisi olla samanlaisessa tilassa kuin alusta aloitettu uusimman version AngularJS-sovellus nykytiedoilla.

Toinen vaihe on uudemman Angularin käyttöönotto ja varsinainen sovelluksen osien päivitys. Tämä vaihe vaatii hyvän Angular-osaamisen, mutta myös ymmärrystä vanhemmista ratkaisuista. Kolmannen osapuolen pakettien päivitys on myös tärkeä osa

tätä vaihetta. Tietämys näiden käsittelystä ja asennuksesta on eduksi tässä vaiheessa päivitystä. Ensimmäisessä vaiheessa käytetty aika helpottaa huomattavasti tätä vaihetta, sillä silloin tässä kohdassa ei tarvitse enää päivitellä vanhaa koodipohjaa.

Miten päivittämisestä voi tehdä mahdollisimman helppoa?

Päivittäminen on monimutkainen prosessi. Tämän prosessin huolellinen suunnittelu helpottaa päivittämistä, kun tiedetään, miten varsinaista päivitystä lähdetään valmistelemaan ja myöhemmin toteuttamaan. Päivittämisen avuksi löytyy useita työkaluja, esimerkiksi Angular-kehittäjäjoukon julkaisema työkalu nimeltään Angular Migration Assistant. Tätä työkalua voidaan käyttää suunnittelun apuna analysoinnissa, onko AngularJS-sovelluksen rakenne valmis sovelluksen päivitykseen. Työkalu tarjoaa myös ehdotuksia sovelluksen rakenteesta. Näiden ehdotusten avulla saadaan tietoa, mitä sovelluksesta tarvitsee vielä muokata ennen varsinaisen päivityksen aloittamista.

Päivittämisprosessiin voi myös tutustua useiden eri verkkokurssien avulla, joiden tarkoituksena on tutustuttaa esimerkkien avulla päivitykseen. Tämän lisäksi AngularJS- ja Angular-osaamista tarvitaan tiimiin päivityksen helpottamiseksi. Kun sovelluksen osien päivittäminen on aloitettu, tulee prosessissa paljon toistoa. Toiston avulla prosessi helpottuu, sillä tietämys sovelluksen osien päivityksestä kertaantuu jokaisella kerralla.

8 Pohdinta

Tämän opinnäytetyön tarkoituksena oli löytää toimeksiantajayritykselle mahdollisimman helppo keino päivittää olemassa oleva AngularJS-sovellus Angular-sovellukseksi. Jokainen AngularJS-sovellus on kuitenkin erilainen ja tämä hankaloittaa päivityksen tekemistä ja tähän soveltuvien ohjeiden löytämistä. Pitkään kehityksessä olevan sovelluksen koodipohja kasvaa, joka tekee päivittämisestä huomattavasti vaikeampaa. Päivittäminen on paljon helpompaa ja nopeampaa, jos sovelluksessa on seurattu tarkasti hyviä kehityskäytänteitä ja otettu käyttöön esimerkiksi TypeScript ja Webpack jo ennen päivityksen suunnittelua. Tällöin työskentely TypeScriptillä ja Webpackin

konfiguroinnin kanssa on tutumpaa. Tämä vähentää päivityksen työmäärää, sillä nämä voidaan ottaa suoraan päivitetyn sovelluksen käyttöön.

AngularJS-sovellukset voivat olla todella laajoja, eikä ne välttämättä muistuta rakenteeltaan uudempaa Angularia. Tässä tapauksessa päivittämiseen tarvittava työmäärä on niin suuri, että sovelluskehityksen päivittäminen ei todennäköisesti ole järkevää. Jos sovellus muistuttaa rakenteeltaan uudempaa Angularia ja riippuvuuksia epäsoiviin kolmannen osapuolen paketteihin ei ole, voi päivittäminen olla järkevää suhteessa vaadittavaan työmäärään.

Ainoastaan tämän tutkimuksen teoriaosuuden perusteella ja kokemuksella tätä päivitystä ei ole mahdollista tehdä. Tämä vaatii paljon pohjatietoa molemmista sovelluskehityksistä, sillä päivitysprosessi on monimutkainen. Tämän tutkimuksen tuloksia ja teoriaosiota voidaan hyödyntää myös perehdyttämistarkoituksessa sovelluskehittäjille. Tämä tutkimus sisältää hyödyllistä tietoa erityisesti kehittäjille, jotka ovat siirtymässä AngularJS-sovelluksesta kehittämään Angular-sovellusta. Näiden sovelluskehitysten välillä olevia eroavaisuuksia on käsitelty tässä tutkimuksessa, joka auttaa hahmottamaan millä tavalla ennestään tutut asiat ovat muuttuneet uudessa sovelluskehityksessä.

Tulosten luotettavuus

Tutkimuksessa on käytetty paljon Angularin virallista dokumentaatiota lähteenä teoriaosiossa. Osa teoriaosiesta pohjautuu kehittäjäjoukon jäsenen kirjoittamaan kirjaan. Lähteenä tämä on luotettava, sillä tämän kirjan julkaisua on edeltänyt tarkastukset ja oikoluku. Tämän kirjan tietoa on myös täydennetty verkosta löytyvillä lähteillä, kuten virallisella dokumentaatiolla. Näiden lisäksi lähteenä tässä tutkimuksessa oli myös ng-conf-tapahtuman esitysten tallenteet. Ng-conf-tapahtuma on maailman suurin Angular-kehittäjille tarkoitettu tapahtuma, jossa asiantuntijat jakavat tietoaan Angulariin liittyvistä aiheista. Tapahtumassa on paljon puhujia, jotka ovat olleet Angularin kehityksessä mukana alusta alkaen.

Tämän tutkimuksen tulokset eivät ole yleistettävissä, sillä jokainen AngularJS-sovellus on erilainen toisistaan ja niissä käytetyt tyylit tehdä sovellus voivat erota toisistaan

huomattavasti. Tämän tutkimuksen teoriaosiota voidaan kuitenkin hyödyntää myös muiden sovelluksien kohdalla.

Mahdolliset kohteet jatkotutkimukselle

Tutkimuksen aineiston keräämisen aikana esille tuli useaan kertaan ohjeistuksia päivittää Angular- tai AngularJS-sovelluskehys myös muihin mahdollisiin tunnettuihin sovelluskehysiin, kuten React. Angular tarjoaa enemmän valmiina kuin useimmat muut sovelluskehukset ja pakottaa tietyntylaiseen koodin kirjoitukseen, joka vähentää joustavuutta. Tässä tutkimuksessa rajattiin tarkoituksella nämä muut vaihtoehdot pois.

Toinen tutkimuksen aikana esiin tullut mahdollinen jatkotutkimuksen kohde on vanhempien Angular-sovelluksien päivitys uudempaan Angular-versioon. Tähän aiheeseen liittyen myös tutkimuskohteena voisi olla Angular CLI:n mukaan ottaminen sovellukseen, jossa sitä ei vielä ole käytössä. Version päivitykseen löytyy runsaasti materiaalia verkosta ja Angular CLI mahdollistaa pakettien päivityksen ”ng update”-komennolla.

Lähteet

\$resource. N.d. AngularJS virallinen dokumentaatio. Viitattu 24.6.2019.
[https://docs.angularjs.org/api/ngResource/service/\\$resource](https://docs.angularjs.org/api/ngResource/service/$resource).

Ahmed, K. 2018. Learn to become a modern Frontend Developer in 2019. Artikkelin Medium-verkkosivulla. Viitattu 11.8. 2019.
<https://medium.com/@kamranahmedse/modern-frontend-developer-in-2018-4c2072fa2b9c>.

AngularJS to Angular Concepts: Quick Reference. N.d. Angularin virallinen dokumentaatio. Viitattu 24.7.2019. <https://angular.io/guide/ajs-quick-reference>.

Attribute Directives. N.d. Angularin virallinen dokumentaatio. Viitattu 9.7.2019.
<https://angular.io/guide/attribute-directives>.

Black, N. 2014. An AngularJS Style Guide and Best Practice for App Structure. Artikkelin virallisessa AngularJS-blogissa. Viitattu 24.6.2019.
<http://blog.angularjs.org/2014/02/an-angularjs-style-guide-and-best.html>.

CLI Overview and Command Reference. N.d. Angularin virallinen dokumentaatio. Viitattu 3.6.2019. <https://angular.io/cli>.

Creating Custom Directives. N.d. Virallinen AngularJS-dokumentaatio. Viitattu 30.5.2019. <https://docs.angularjs.org/guide/directive>.

Darwin, P. B. 2018. Stable AngularJS and Long Term Support. Artikkelin Angular Blog-verkkosivulla. Viitattu 9.4.2019. <https://blog.angular.io/stable-angularjs-and-long-term-support-7e077635ee9c>.

Frequently Used Modules. N.d. Angularin virallinen dokumentaatio. Viitattu 8.7.2019.
<https://angular.io/guide/frequent-ngmodules>.

Hevery, M. & Green, B. 2014. Keynote – NG-Conf 2014. Videotallenne ng-conf 2014-tapahtumasta YouTube-verkkosivulla. Viitattu 27.5.2019.
<https://youtu.be/r1A1VR0ibiQ>.

Introduction to modules. N.d. Angularin virallinen dokumentaatio. Viitattu 8.7.2019.
<https://angular.io/guide/architecture-modules>.

Introduction to services and dependency injection. N.d. Angularin virallinen dokumentaatio. Viitattu 13.7.2019. <https://angular.io/guide/architecture-services>.

Kananen, J. 2012. Kehittämistutkimus opinnäytetyönä: Kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Modules. N.d. Virallinen AngularJS-dokumentaatio. Viitattu 28.5.2019.
<https://docs.angularjs.org/guide/module>.

Neale, A. 2016. An Introduction To Gulp. Artikkele Medium-verkkosivulla. Viitattu 25.7.2019. <https://medium.com/@andy.neale/an-introduction-to-gulp-808465e7fded>.

No FilterPipe or OrderByPipe. N.d. Angularin virallinen dokumentaatio. Viitattu 3.6.2019. <https://angular.io/guide/pipes#no-filter-pipe>.

Olson, E. 2018. 2 New Tools to help with AngularJS to Angular Migrations. Artikkele Angular Blog-verkkosivulla. Viitattu 1.4.2019. <https://blog.angular.io/migrating-to-angular-fc9618d6fb04>.

Papa, J. & Wahlin, D. 2017. Diving into TypeScript. Videotallenne ng-conf-tapahtumasta. Viitattu 8.7.2019. <https://youtu.be/4xScMnaasG0>.

Papa, J. N.d. Angular 1 Style Guide. Github. Viitattu 18.6.2019. <https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md>.

Pipes. N.d. Angularin virallinen dokumentaatio. Viitattu 9.7.2019. <https://angular.io/guide/pipes>.

Poshtaruk, A. 2019. Promises vs Observables for AngularJS-to-Angular migration. Artikkele itnext-verkkosivulla. Viitattu 30.8.2019. <https://itnext.io/promises-vs-observables-for-angularjs-to-angular-migration-1161afacef7e>.

Routing & Navigation. N.d. Angularin virallinen dokumentaatio. Viitattu 15.7.2019. <https://angular.io/guide/router>.

Seshadri, S. & Green, B. 2014. AngularJS Up & Running: Enhanced productivity with structured web apps. Sebastopol: O'Reilly Media.

Stankiewicz, A. 2017. How to migrate away from Bower? Artikkele bowerin virallisessa blogissa. Viitattu 18.6.2019. <https://bower.io/blog/2017/how-to-migrate-away-from-bower/>.

Structural Directives. N.d. Angularin virallinen dokumentaatio. Viitattu 9.7.2019. <https://angular.io/guide/structural-directives>.

Style Guide. N.d. Angularin virallinen dokumentaatio. Viitattu 26.7.2019. <https://angular.io/guide/styleguide>.

Upgrading from AngularJS to Angular. N.d. Angularin virallinen dokumentaatio. Viitattu 1.4.2019. <https://angular.io/guide/upgrade>.

What is npm? 2011. Artikkele Node.js-verkkosivulla. Viitattu 25.7.2019. <https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/>.

Wilken, J. 2016. A Guide to Building Quality Angular 1.5 Components. Artikkele Sitepoint-verkkosivulla. Viitattu 2.6.2019. <https://www.sitepoint.com/building-angular-1-5-components/>.

Vilkka, H. 2015. Tutki ja kehitä. Jyväskylä: PS-kustannus.

Liitteet

Liite 1. Step-by-step ohje AngularJS => Angular

AngularJS => Angular päivitysohje step-by-step

1. Koodipohjan valmistelu tyylioppaan mukaiseksi.

Tärkein kohta tyylioppaassa on "yhden sääntö", eli jokainen controller, service ja myöhemmin komponentti ovat omassa tiedostossaan. Tiedostojen nimestä tulisi myös nähdä heti sisältääkö tiedosto käsittelijän vai palvelun, esimerkiksi people-list.controller.js

2. Bower vaihto npm

Vanhentunut bower-pakettienhallinta voidaan vaihtaa npm kopioimalla dependencies-listaus package.json-tiedostoon bower.json-tiedostosta. Tässä kohdassa täytyy ottaa huomioon, että versionumerot eivät välttämättä täsmää täysin npm ja bower välillä. Paketit package.json-tiedostossa (npm) eivät tue isoja kirjaimia kuten bower, joten vastaavat paketit tulee etsiä npm-pakettienhallinnan sivuilta npmjs.com.

3. TypeScript käyttöönotto

TypeScript asennetaan devDependencies-kohtaan ajamalla komento "npm install typescript --save-dev". Tämä jälkeen src-kansion .js-tiedostot nimetään uudelleen .ts-päätteellä. Asennuksen mukana on TypeScript-kääntäjä ja "tsc"-komennon jälkeen kaikki toimii taas kuten ennenkin luomalla vastaavat .js-päätteiset tiedostot.

4. Moduulien lataaja

Jotta TypeScriptiä voidaan hyödyntää, tarvitsee käyttöön ottaa moduulien lataaja. Paras vaihtoehto tähän on webpack. Webpackin lisäksi asennetaan sille lisäosia erilaisten tehtävien suorittamiseen. Webpackin konfigurointi tapahtuu webpack.config.js-nimisessä tiedostossa.

5. **AngularJS-version päivitys**

Päivitetään AngularJS-versio vähintään 1.5.x, jolloin voidaan luoda komponentteja. Versio kannattaa kuitenkin päivittää mahdollisimman tuoreeksi, sillä mitä uudempi versio, sen lähempänä ollaan uudempaa Angularia. Version päivityksessä tulee ottaa huomioon AngularJS-dokumentaatiosta löytyvät "breaking changes"-muutokset, ja korjata ne tässä vaiheessa.

6. **AngularJS-komponenttirakenne**

Jotta AngularJS-komponentteja voidaan käyttää yhdessä uudemman Angularin kanssa, tarvitsee niiden noudattaa versiossa 1.5.x mukana tullutta komponenttirajapintaa. Aikaisemmin tämä on toiminut controllerien ja direktiivien avulla. Komponenttirakenteen myötä \$scope-palvelua ei enää tarvita, sillä komponentit käyttävät controllerAs-syntaksia. \$scope-palvelun sijasta voidaan käyttää this-avainsanaa.

7. **Uuden Angularin asennus**

Uusi Angular asennetaan lisäämällä tarvittavat paketit package.json-tiedostoon ja ajamalla komennon "npm install". Tämän lisäksi luodaan polyfills.ts-tiedosto, johon määritellään uudempien selaimien natiivi toimintoja, joita vanhemmat selaimet eivät vielä tue. Tämä tiedosto importoidaan main.ts-tiedostossa.

8. **Sovelluksen ajaminen hybridinä (AngularJS + Angular)**

Seuraavaksi poistetaan ng-app-direktiivi index.html-tiedostosta. Tämä korvataan main.ts-tiedostoon lisättävällä upgradeModulen tarjoamalla bootstrap-metodilla. Main.ts-tiedostoon määritellään myös kaikki moduulit mitä käytetään, esimerkiksi UpgradeModule.

9. **Palveluiden päivitys Angular-muotoon**

Seuraavaksi voidaan aloittaa palveluiden päivitys uudemman Angularin muotoon. Tässä kohdassa päivitetään AngularJS:n valmiit palvelut vastaamaan Angularia, esimerkiksi \$http korvataan HttpClientilla. HttpClient palauttaa observablen, joka voidaan muuttaa promiseksi rxjs:n tarjoamalla toPromise()-metodilla, jolloin komponenttien logiikkaa ei tarvitse muuttaa. Jotta palvelua voidaan käyttää AngularJS-komponenttien kanssa, tarvitsee se alentaa käyttäen downgradeInjectable()-metodia.

10. Komponenttien päivitys Angular-muotoon

Angular komponentit voidaan alentaa AngularJS-yhteensopiviksi `downgradeComponent()`-metodilla. Tässä on tärkeää tiedostaa, että `downgradeComponent()`-metodia varten luodaan AngularJS-direktiivi. Mallipohjan kirjoitustyyli eroaa myös, joten esimerkiksi `ng-repeat`-direktiivi tulisi korvata `*ngFor`-direktiivillä. Myös esimerkiksi kuvien `src`-attribuutti vaatii hakasulkeet `[src]` toimiakseen. AngularJS-filttereitä ei voida suoraan päivittää Angularin vastaaviksi, vaan ne pitää kirjoittaa kokonaan uudelleen.

11. Reitittimen päivitys Angular-muotoon

Korvataan `ui-router` Angularin mukana tulevalla reitittimellä. Tämän reitittimen toimimiseksi `index.html`-tiedostoon tarvitsee määritellä `<base href="/">` tagi `head`-osion alkuun. Mallipohjassa `ui-view`-direktiivi korvataan `<router-outlet>`-komponentilla ja `ui-sref`-määrittelyn tilalle `[routerLink]`. TypeScript koodissa `$state.go` tilalle vaihdetaan reitittimen `.navigate`-kutsu. Reitittimen parametrit voidaan hakea tarkkailijamuodossa `params.subscribe`-kutsua hyödyntäen.

12. AngularJS poistaminen

Poistetaan TypeScript-koodista `import`-määrittelyt liittyen AngularJS-paketteihin. Poistetaan nämä paketit myös `package.json`-tiedostosta. Myös kaikki AngularJS-pohjaiset moduulit juurimoduuli mukaan lukien poistetaan. Määrittelyt, joissa komponentteja tai palveluita alennetaan tai ylennetään, voidaan poistaa. Myös Angularin Upgrade-moduulin viittaukset tulee poistaa koodista ja `package.json`-tiedostosta.