Jyri Högman

ANGULAR 2 -MIGRAATIO

Tietotekniikan koulutusohjelma

2017

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

ANGULAR 2 -MIGRAATIO

Högman, Jyri
Satakunnan ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Tammikuu 2017
Ohjaaja: Auramo, Yrjö
Sivumäärä: 29
Liitteitä: 0

Asiasanat: migraatio, komponentti, riippuvuus, kirjasto

Opinnäytetyön aiheena oli Angular 2 -migraatio. Tutkimuksen tarkoitus oli luoda migraatiomenetelmä, jonka avulla toimeksiantajayritys pystyy päivittämään käyttöliittymäpuolen Angular 1 -ohjelmistokoodin versioon 2. Menetelmän luomiseksi haettiin vastauksia seuraaviin kysymyksiin: Mitä hyötyjä ja mahdollisia haittoja migraatiosta voisi seurata? Miten migraatio onnistuu sovellukseen? Miten se tulisi suorittaa ja kuinka työläs migraatio tulisi olemaan?

Migraatiomenetelmän aikaansaamiseksi suoritettiin ensin koemigraatio aiemmin tehtyyn ajanvarausjärjestelmä-sovellukseen, joka noudattaa samaa tyyliopasta kuin toimeksiantajayrityksen ohjelmisto. Koemigraatioista saatujen havaintojen pohjalta suoritettiin toimeksiantajayrityksen ohjelmiston yksittäisten osien päivittäminen Angular 2:en.

Tutkimustyön perusteella tultiin tulokseen, että Angular 1 -sovellus voidaan täysin uudelleenkirjoittaa sisältämään vain Angular 2 -koodia, tai migroida komponentti kerrallaan käyttämällä Angular 2:en UpgradeAdapter moduulia. Molempia työtapoja tutkittiin, ja tultiin siihen tulokseen, että erityisesti koodimassaltaan pieneen sovellukseen kannattaa migraatio suorittaa kirjoittamalla se kokonaan uudestaan. Suuressa sovelluksessa tilanteeseen vaikuttaa erityisesti se, millä Angular 1:en versiolla sovellus on toteutettu. Angular 1.5 component API:a noudattavat sovellukset on vaivattomampia kääntää Angular 2:lle, kuin aikaisemmat Angular 1:en versiot. Muita vaikuttavia tekijöitä ovat sovelluksen komponenttien mahdolliset riippuvuudet ja kolmannen osapuolen kirjastot. Jos komponentit ja palvelut sisältävät useita riippuvuuksia, voi UpgradeAdapterin käyttö olla hankalaa ja välttämättä kaikkia komponentteja ei voi käyttää sillä suoraan.

Migraatiotutkimuksessa todettiin, että koodimassaltaan pienikokoiselle sovellukselle tehtynä migraatio onnistuu helpoiten. Sovelluksen koon kasvaessa migraation tekeminen vaikeutuu ja työtavan valitsemisen tulee olla tarkkaan mietitty.

ANGULAR 2 MIGRATION

Högman, Jyri
Satakunta University of Applied Sciences
Degree Programme in Computer Science
January 2017
Supervisor: Auramo, Yrjö
Number of pages: 29
Appendices: 0

Keywords: migration, component, dependency, library

The subject of the research was Angular 2 migration. Purpose of the thesis was to get a migration method that the company can use in the future, when migrating from Angular 1 to Angular 2. To develop the method, main questions were: What advantages and disadvantages would the migration have? How can the migration be done for the company's application? How should the migration be conducted?

To achieve the method, a test migration was initially performed for a reservation application I had previously made. After that the migration was started on a part of the company's application.

There are two ways to complete the migration, a complete rewrite to Angular 2 or rewrite one component at a time by using the Angular 2 UpgradeAdapter module. The research results show that the level of effort required for the migration depends highly on your Angular 1 codebase. What matters the most is which Angular 1 version is used in the application. If the application is written following the Angular 1.5 component API, the migration is the easiest. Multiple components and services depending on each other makes the migration harder if the application is developed with UpgradeAdapter. Depending on the Angular 1 code, some components or services might not be upgradable using the UpgradeAdapter module in Angular 2.

Results from the migration research showed that a small-scale application is the easiest to migrate. The larger the application is the harder it will be harder to migrate and the method of choice should be carefully considered.

LYHENTEET JA KÄSITTEET

HTTP	Tiedonsiirtoon käytettävä protokolla (Hypertext Transfer Protocol).
DOM	Ohjelmointirajapinta HTML, XML ja SVG dokumenteille (Document Object Model).
HTML	Merkintäkieli, jolla rakennetaan ja näytetään tietoa internetsivulla (Hypertext Markup Language).
AJAX	Joukko tekniikoita, joilla selainsovelluksista tehdään dynaamisempia (Asynchronous Javascript And XML).
CSS	Tyyliohje WWW-dokumenteille (Cascading Style Sheets).
JavaScript	Web-ympäristössä käytettävä dynaaminen komentosarjakieli.

SISÄLLYSLUETTELO

1 JOHDANTO

Web-ohjelmoinnin tavat ja tekniikat kehittyvät jatkuvasti. Erityisesti viime vuosina JavaScriptiin on tullut uusia ohjelmistokehyksiä ja kirjastoja. Ohjelmistokehyksien on tarkoitus helpottaa ohjelmoijan työtä (Ivanovs 2016). Tämä opinnäytetyö pohjautuu JavaScript-ohjelmointikielen AngularJS-ohjelmistokehykseen. Tutkimus perehtyy AngularJS:n versiomigraatioon, ja työn lopputuloksena on toimeksiantajayrityksen tarpeita vastaava migraatiomenetelmä, jolla AngularJS 1 voidaan päivittää versioon 2.

Menetelmän aikaansaamiseksi suoritettiin ensin koemigraatio aiemmin tehtyyn ajanvarausjärjestelmä-sovellukseen, jonka jälkeen suoritetaan toimeksiantajayrityksen ohjelmiston erinäisten osien päivittäminen Angular 2:een. Toimeksiantajayrityksen ohjelmisto ja ajanvarausjärjestelmä noudattavat samaa Angular 1 -tyyliopasta, joten ajanvarausjärjestelmän migraatiomenetelmää voidaan hyödyntää, kun migraatiotutkimus tehdään yrityksen sovellukseen. Valmiit migraatiomenetelmät tulevat kuitenkin olemaan hyvin erilaiset. Ajanvarausjärjestelmä on ohjelmistona pieni, kun taas yrityksen ohjelmisto koostuu useista alijärjestelmistä, joiden pareissa työskentelee satoja kehittäjiä. Ohjelmistokoodin rivimäärissä testisovellus on satojen koodirivien luokkaa, kun taas isommat ohjelmistoprojektit saattavat sisältää miljoonia rivejä koodia.

Yhtenä kysymyksenä päivityksessä tulee olemaan toteutuksen tyyli. Onko Angular 2 -sovelluksen tarkoitus olla asiakaskäytössä samaan aikaan, kun Angular 1 -sovelluksen? Tähän tarkoitukseen sopiva vaihtoehto olisi hybridiapplikaatio, jossa ajetaan samanaikaisesti kumpaakin Angularin versiota. Toinen vaihtoehto on aloittaa kehitys alusta uutta versiota varten ja suorittaa kehitys alusta loppuun ennen tuotantoon viemistä.

Opinnäytetyön aihetta koskevia lähteitä löytyy erityisesti internetistä. Angularin oma dokumentaatio on kattava ja erilaiset ominaisuudet on listattu selkeästi. Dokumentaatioon viitataan kuvaillessa Angular 2:n ominaisuuksia. Muut lähteet ovat

blogien, e-kirjojen ja kirjojen muodossa. Lähteet on valittu ominaisuuksiensa takia aihekohtaisesti. Web-ohjelmointiin kuuluu modernisti vapaamuotoinen dokumentaatio. Blogikirjoituksien sisältöä Angular 2:n osalta on verrattu Angularin omaan dokumentaatioon ja varmistettu, että kirjoitus on viimeisen julkaisukandidaatin tai virallisen version tasolla. Näin ristiriidassa olevat blogikirjoitukset on eliminoitu lähteistä laadun takaamiseksi.

Opinnäytetyön kappaleessa kaksi esitellään selainsovelluksia yleisesti. Kappaleessa kolme käsitellään Angular -ohjelmistokehyksen 1 ja 2 versioita. Opinnäytetyön tutkimuksen tarkoitus ja tutkimusmenetelmä esitellään tarkemmin kappaleessa neljä. Varsinainen tutkimus ja sen vaiheet käydään läpi kappaleessa viisi. Tutkimuksen tuloksia analysoidaan kappaleessa kuusi. Työ pyrittiin pitämään tarpeeksi suppeana rajaamalla se Angularin migraatioprosessiin ja Web-kehitykseen kahden eri Angularin version näkökulmasta. Lukijalta suositellaan kokemusta JavaScript -ohjelmoinnista ja yksisivuisista sovelluksista (Single Page Application).

2 SELAINSOVELLUKSET

Ohjelmistoalalla on tapahtumassa muutoksia ja selaimia sisältävien mobiililaitteiden suosio on kasvanut valtavasti. Natiivien sovellusten tuottaminen eri alustoille voi olla hankalaa, mutta tähän löytyy ratkaisu, HTML-selainsovellus. Selainsovellukset ovat helposti asiakkaiden ja ihmisten saatavilla, riippumatta laitteesta. HTML5-sovellus toimii selaimen moottorilla ja käyttää Web-tekniikoita, mutta muistuttaa enemmän sovellusta kuin verkkosivua. HTML-sovellukseen voidaan ottaa mukaan avoimia tekniikoita, kuten CSS ja JavaScript, joilla voidaan tuoda sovellukseen parempaa ulkonäköä ja erilaisia toimintoja. Selainsovellus ei ole joukko linkitettyjä verkkosivuja, vaan se on yksi toiminnallinen kokonaisuus. Sovellus ei ole riippuvainen esimerkiksi selaimen navigointitoiminnoista. Sen ohjaustoimintoihin liittyvät yleensä esimerkiksi painikkeet, painikkeen painallus tai valikot. (Lehdonvirta & Korpela 2015, 13-14.)

Tahtotila kehittäessä selainsovellusta on usein laiteriippumattomuus. Laiteriippumattomuus tarkoittaa sitä, että sama ohjelma ja koodi toimivat eri laitteilla. HTML5-sovellusalusta luo tälle mahdollisuudet, mutta toteutukseen vaaditaan laadukasta ohjelmakoodia. Selainsovelluksessa ei ole pelkästään kyse laiteriippumattomuudesta, vaan myös yleisestä toimivuudesta ja käyttäjäkokemuksesta. Sovelluksen responsiivisuus esimerkiksi näytön koon mukaan on yksi perusasia, mitä laadukkaalta selainsovellukselta toivotaan. Näytöllä näkyvä sovellus voi näin olla aseteltu erilaiseksi, mutta sisältö on yleensä sama. Selainsovellusten vahvuutena on myös vapaus kehitysympäristöä valittaessa. Sovelluksen voi ohjelmoida esimerkiksi Windowsin Muistio-ohjelmalla. Suositeltavaa on kuitenkin käyttää tarkoitukseen kehitettyä kehitysympäristöä. (Lehdonvirta & Korpela 2015, 15-16.)

Laiteriippumattomuudessa ei ole kyse pelkästään näytön koosta. Laitteen suorituskyky, tekstin syöttö ja käyttäjän painallukset ovat erilaisia laitteesta riippuen. Tekstin syöttö vaihtelee näppäimistöstä kosketusnäyttöön, samoin painallukset. Oli käytössä sitten näppäimistö tai kosketusnäyttö, pitää käyttäjäkokemus olla miellyttävä. Web-sovellus tulisi suunnitella yleisesti kaikille laitteille ja tarvittaessa vain lisätä tiettyjä ominaisuuksia jollekin laitetyypille tarvittaessa. Tällä hetkellä esimerkiksi

tekniikka puheen tunnistamiseen ja syntetisointiin on jo olemassa. Tekniikan parantuessa voi siitä tulla keskeinen osa Web-sovelluksia. Kehitys luo jatkuvasti ongelmia käyttöliittymille ja käyttäjän ja sovelluksen vuorovaikutukselle. (Lehdonvirta & Korpela 2015, 24-26.)

Selainsovelluksien toiminnallisuudesta vastaa usein ohjelmointikieli JavaScript, jolla toteutetaan sovelluksen toiminnan olennainen osa. JavaScript on tullut käyttöön vuonna 1995 (Hamilton 2008). Opinnäytetyössäni käsitellään Googlen kehittämää Angularia, joka on JavaScriptin ohjelmistokehys. JavaScriptin modulaarisuus on helpottanut sovelluksien kehittämistä. Kun puhumme modulaarisesta sovelluksesta, tarkoitamme yleensä sitä, että sovellus koostuu tarkkaan erotelluista osista ohjelmakoodia. Nämä osat ohjelmakoodia voidaan ajatella moduuleiksi. Kun sovelluksen eri osat ovat eroteltu esimerkiksi käyttötarkoituksen mukaan toisistaan, vähenee moduulien riippuvuuksien määrä. Moduuleita tässä tarkoituksessa ajatella myös uudelleenkäytettäviksi komponenteiksi, ne ovat osia ohjelmaa, jotka tarpeeksi eroteltuina toisistaan ovat uudelleenkäytettäviä. (Osmani 2012)

JavaScriptissa moduulien tyyppi vaikuttaa siihen, miten moduulit luodaan ja millä niitä voidaan tuoda sovellukseen. Käsittelen opinnäytetyössäni CommonJS ja AMD moduulityyppejä. CommonJS moduulit määritetään module.exports funktiolla, ja tuodaan sovellukseen require funktiolla. Kuvassa yksi määritellään myModule niminen CommonJS moduuli. Määrittämisen jälkeen CommonJS -moduulityyppi tietää mitä tuoda tästä moduulista riippuviin muihin moduuleihin. (Kasireddy 2016)

```
1   function myModule() {
2     this.hello = function() {
3       return 'hello!';
4     }
5
6     this.goodbye = function() {
7       return 'goodbye!';
8     }
9   }
10
11  module.exports = myModule;
```

Kuva 1. CommonJS moduulin määrittäminen (Kasireddy 2016)

Jonkin toisen sovelluksen moduulin tarvitessa määritettyä myModule:a, käytämme require funktiota riippuvuuden tuomiseen, kuten kuvassa kaksi. CommonJS on kehitetty erityisesti palvelinpuolen modulaarisen JavaScriptiin. Sillä ladataan moduulit yksitellen synkronisesti. Synkroninen lataus on tuonut aiemmin ongelmia käyttöliittymäohjelmointiin. (Kasireddy 2016)

```
var myModule = require('myModule');

var myModuleInstance = new myModule();
myModuleInstance.hello(); // 'hello!'
myModuleInstance.goodbye(); // 'goodbye!'
```

Kuva 2. myModule moduulin tuonti require funktiolla (Kasireddy 2016)

AMD eli Asynchronous Module Definition -moduulityyppi on vastaus niille, jotka ajattelevat CommonJS moduulijärjestelmän olevan keskeneräinen selaimelle sen synkronisen latauksen takia. AMD spesifioi standardin modulaariselle JavaScriptille niin, että moduulit voivat ladata riippuvuutensa asynkronisesti. (Kasireddy 2016)

AMD määrittää ja lataa moduulit define() funktiolla. Funktio on muotoa define(id?: String, riippuvuudet?: String[], factory: Function|Object);. Definelle siis annetaan argumentteina moduulin nimi merkkijonona, merkkijonokokoelma riippuvuuksista ja moduulin määrittely. Määrittely voi olla funktio tai objekti. Kuvassa kolme on esimerkki siitä, miten aiemmin jo määritelty myModule määriteltäisiin AMD moduulina. Jos moduulin tarvitsee ladata joku toinen moduuli, lisätään riippuvuudet define funktion kokoelmaan. (Osmani 2012)

```
define([], function() {

  return {
    hello: function() {
      console.log('hello');
    },
    goodbye: function() {
      console.log('goodbye');
    }
  };
});
```

Kuva 3. AMD moduulin määrittely (Kasireddy 2016).

3 ANGULAR

3.1 AngularJS (Angular 1)

AngularJS on rakenteellinen, Googlen kehittämä, ohjelmistokehys dynaamisille Web-sovelluksille. AngularJS antaa ohjelmoijalle mahdollisuuden käyttää HTML-kieltä sovelluksen mallipohjan kielenä ja laajentaa HTML:n syntaksia ilmaisemaan sovelluksen komponentteja selkeästi ja tehokkaasti. Angularin datan sitominen ja riippuvuuksien injektointi eliminoi tarpeen kirjoittaa suurta määrää koodia itse. Angular toimii selaimessa, mikä tekee siitä hyvän yhdistelmän kaikkien palvelinpuolen teknologioiden kanssa. Angular pyrkii minimoimaan erot dokumenttikeskeisen HTML:n ja Web-sovelluksen välillä tekemällä omia HTML-rakennelmia. Angular opettaa selaimelle uutta syntaksia käyttämällä direktiivi-rakennelmaa. (Google [1] 2016)

Angular on suunniteltu erityisesti käytettäväksi CRUD-sovelluksissa, joita suurin osa Web-sovelluksista nykyään on. Ohjelmat, joissa etsitään, näytetään, luodaan, poistetaan ja päivitetään jotain tiettyä dataa sopivat hyvin Angularilla toteutettaviksi sovelluksiksi. Angular ei ole pieni osa asiakaspuolen ratkaisua, vaan sillä pystytään toteuttamaan ratkaisu kokonaan. Se hoitaa DOM:in ja AJAX:n yhdistämisen ohjelmakoodilla itse ja asettaa sille hyvin määritellyn rakenteen. Käyttämällä Angularia pelkän JavaScriptin sijaan säästytään esimerkiksi ohjelmalliselta DOM:n manipuloinnilta ja vastakutsujen rekisteröinneiltä. Lisäksi datan siirtely palvelimen ja käyttöliittymän välillä helpottuu. Angular ei kuitenkaan sovellu kaikkiin Web-käyttöliittymiin. Esimerkiksi pelit ja jotkin graafiset käyttöliittymät, jotka vaativat monimutkaista DOM:n manipulointia eivät ole suunniteltu tehtäväksi Angularilla. (Google [1] 2016)

Angularin aikaisempaa versiota aloitettiin kirjoittaa puhtaasti JavaScriptillä, mutta tällä hetkellä on suositeltua käyttää TypeScriptiä. TypeScript laajentaa JavaScriptin ominaisuuksia ja kääntyy JavaScriptiksi, jota kaikki selaimet ymmärtävät. Esimerkiksi Angular 2:n kehittäjä Victor Savkin sanoo, että pelkästään TypeScriptin paremman

navigaation, automaattisen täydennyksen ja refaktoroinnin ansioista kannattaa siirtyä JavaScriptistä TypeScriptiin. (Savkin 2016)

Angular 1 -sovelluksen rakenneosia ovat uusimmissa ohjelmistokehyksen versioissa komponentit, mutta ennen 1.5 versiota Angular 1 -sovellukset koostuivat kontrollereista ja direktiiveistä. Angular 1.5 -komponentteja voi ajatella yhteen sulautettuina kontrollereina ja direktiiveinä (Google [3] 2016). Niitä kutsutaan myös komponentti-direktiiveiksi. Angularin kehitystiimi suosittelee kehittäjiä käyttämään Angularin 1.5 Component API:n mukaisia komponentteja, mutta esimerkiksi vanhemmissa sovelluksissa näkyy edelleen direktiivejä ja kontrollereita. (Parviainen 2015)

Jokaisella Angular 1 -applikaatiolla on moduuli. Moduulit sisältävät applikaation kontrollerit, direktiivit ja palvelut. Moduuleilla on myös omat metodinsa itsensä ja esimerkiksi kontrollerien ja palveluiden initialisointiin. Angularin module metodille annetaan kaksi parametria, ensimmäinen on moduulin nimi ja toinen on kokoelma. Kuvassa neljä on esimerkki moduulin luomisesta. Esimerkissä kokoelma on tyhjä, mutta kokoelmaan voidaan asettaa lista merkkijonoja. Merkkijonot ovat moduulien nimiä, joita sovellus tarvitsee toimiakseen. Kokoelma on lista sovelluksen riippuvuuksista. (Cooper 2013)

```
var app = angular.module('myapp', []);
```

Kuva 4. Angular 1 moduuli (Panda 2014).

Angular 1 -kontrollerien avulla HTML-mallipohjan ei tarvitse sisältää bisneslogiikkaa tai toiminnallisuutta. Kontrollerit ovat HTML-mallipohjasta usein eristettynä erilliseen JavaScript-tiedostoon. Kontrolleri luodaan moduulin sisältämällä controller() metodilla. Yleisesti kontrollerien ei tulisi sisältää useita toimintoja, vaan niiden tulisi sisältää vain yhdelle näkymälle tarvittava bisneslogiikka. Kontrollerit pidetään kevyinä kirjoittamalla tarvittavia toimintoja palveluihin ja listaamalla palvelut kontrollereihin riippuvuuksina. (Cooper 2013)

Angularin direktiivit voivat olla HTML-elementtejä, attribuutteja tietyssä elementissä tai elementin luokkia. Direktiivit tuovat toimintoja DOM-elementteihin. Korkealla

tasolla ne ovat merkkejä DOM-elementissä. Angularin HTML-kääntäjä kiinnittää toimintoja kyseiseen DOM-elementtiin, käyttämällä esimerkiksi tapahtumien kuuntelijoita. Angular sisältää sisäänrakennettuja direktiivejä, esimerkiksi ngBind, ngModel ja ngClass, mutta ohjelmoijan on myös mahdollista tehdä omia direktiivejä. Kun Angular alkulataa ohjelman, HTML-kääntäjä tarkastaa vastaako HTML-elementti direktiivin valitsijaa. Samoin kuin kontrollerit, direktiivit luodaan moduuleissa, mutta directive() funktiolla. (Google [1] 2016)

Kuvassa 5 app.directive() funktio rekisteröi uuden direktiivin moduulille. Ensimmäinen argumentti funktiossa on direktiivin nimi, toinen argumentti on funktio, joka palauttaa direktiivin määrittelevän olion. Oliossa voidaan injektoida tarvittavat palvelut, rajoittaa direktiivin käyttö ja asettaa HTML-mallipohja tai sen osoite. Esimerkissä on rajoitukseen asetettu ’AE’, tämä tarkoittaa sitä, että direktiiviä voidaan käyttää HTML-attribuuttina ja elementtinä. (Panda 2014)

```
var app = angular.module('myapp', []);

app.directive('helloWorld', function() {
  return {
      restrict: 'AE',
      replace: 'true',
      template: '<h3>Hello World!!</h3>'
  };
});
```

Kuva 5. Angular 1 direktiivi (Panda 2014).

3.2 Angular 2

Angular 2.0 julkaistiin 15.9.2016, tätä ennen ohjelmistokehyksestä esiintyi julkisesti saatavana julkaisukandidaatteja, sekä beta- ja alpha versioita. Googlen tavoitteena oli saada Angular 2 erityisesti nopeammaksi ja helppokäyttöisemmäksi kuin Angular 1. Angular 2:n oppiminen pitäisikin onnistua nopeammin kuin aikaisemman version. Suorituskyky on Angular 2 -ohjelmistossa nopeampi erityisesti muutosten tarkkailun ja datan sitomistyylin ansiosta. Angular 2 tuo mukanaan myös ohjelmiston parannetun

modulaarisuuden laiskan lataamisen avulla, Angular 1:ssä pystyttiin ohjelmiston osia lataamaan tarvittaessa käyttämällä esimerkiksi ocLazyLoad moduulia. Ominaisuus ei kuitenkaan ole Angular 1:ssä natiivi. Laiska lataaminen tarkoittaa käytännössä sitä, että joku osia sovelluksesta ladataan vasta, kun sovellus niitä tarvitsee. Näin esimerkiksi Web-sovelluksen ensilatauksen koko saadaan pienemmäksi, ja sovellus aukeaa nopeammin. (Angular University 2016)

Angular 2 -sovelluksen osina toimivat komponentit ja moduulit. Ohjelmiston juurena toimii aina juurimoduuli, joka sisältää @NgModule dekoraattorifunktion. @NgModule sisältää metadata olion, joka sisältää tietoja siitä, että miten Angular kääntää ja suorittaa moduulin koodia. Siinä identifioidaan moduulin omat komponentit, direktiivit, putket ja palvelut. Angularin modulaarisuus helpottaa ohjelmakoodin organisoimista ja kolmannen osapuolen komponenttien käyttöä. Monet Angularin kirjastot ovat moduuleja. Esimerkiksi BrowserModule ja RouterModule ovat Angularin ydinmoduuleja, BrowserModulea tarvitaan jokaiseen Angular 2 Web-applikaation rekisteröimään Angularille kriittisiä ominaisuuksia, ja RouterModule on Angularin reitittimen moduuli, jolla sivuille rekisteröidään osoitteet. Angular 2 -sovellus tarvitsee vähintään yhden moduulin, mutta sovelluksen koosta riippuen moduuleita voi olla useita. Laajaan sovellukseen voidaan tehdä toimintokohtaisia moduuleita. Toimintokohtaisia moduuleja voidaan ladata vasta, kun sovellus niitä tarvitsee. Kuvassa 6 on esimerkkimoduuli, jossa sovellukseen tuodaan BrowserModule, ilmoitetaan AppComponent tämän moduulin jäseneksi ja bootstrapataan AppComponent. AppComponentiksi nimetään yleensä sovelluksen juurikomponentti Angular 2 -nimeämiskovention mukaan. (Google [4] 2017)

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import
       { AppComponent }  from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Kuva 6. Sovelluksen esimerkkimoduuli (Google [4] 2017).

Angular -sovellus sisältää vähintään yhden juurimoduulin, mutta se voi myös sisältää useita moduuleja. Komponentti on sovelluksen perus rakenneosa, Angular 2 -sovellus on puu erilaisia Angular 2 -komponentteja. Komponentilla on aina HTML-mallipohja ja komponentti kuuluu aina johonkin NgModuleen. Angular 2 -komponentti merkataan @Component dekoraattorifunktiolla. Funktio sisältää usein ainakin valitsijan, HTML-mallipohjan tai polun mallipohjatiedostoon ja polut tyylitiedostoihin. Kuvassa 7 komponentille määritellään moduleId, selector, templateUrl ja styleUrls. (Google [4] 2017)

```
@Component({
  moduleId: module.id,
  selector: 'movie-list',
  templateUrl: 'movie-list.component.html',
  styleUrls: [ 'movie-list.component.css' ],
})
```

Kuva 7. Sovelluksen komponentti (Google [6] 2017).

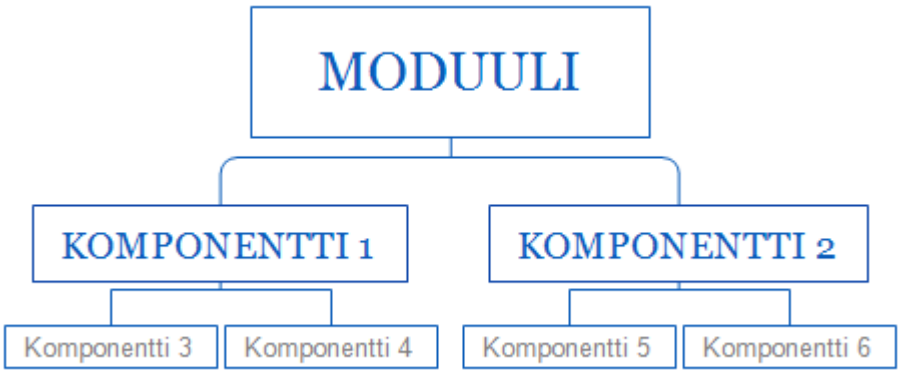
Riippuvuuksien injektointi on myös kokenut suuria muutoksia. Angular 2 -injektointi tapahtuu moduulissa tai komponentin konstruktorissa. Injektoitu riippuvuus on lähes aina palvelu. Angular lukee komponenteissa riippuvuuden konstruktorin parametrista ja sen tyypistä. Parametrin tyyppi toimii injektorin tunnisteena. Kuvassa 8 on esimerkki, miten komponenttiin injektoidaan LoggerService, palvelua käytetään logger parametrilla. (Google [5] 2017)

```
constructor(logger: LoggerService) {
  logger.logInfo('Creating HeroBiosComponent');
}
```

Kuva 8. Riippuvuuden injektointi komponentissa (Google [5] 2017).

Angular 2 -riippuvuudet pitää rekisteröidä aina komponentin tai moduulin metadatan providers-kokoelmaan. Riippuen siitä, missä osassa ohjelmaa rekisteröinti tehdään, näkyy rekisteröity riippuvuus sen komponentin tai moduulin lapsikomponenteille ja moduuleille. Jos esimerkiksi Kuvan 9 mukaan LoggerService rekisteröidään

tarjoajaksi moduulin providers-kokoelmassa, näkyy LoggerServicen sama instanssi moduulin jokaisessa komponentissa. Jos rekisteröinti tehdään Komponentti 1:ssä, voidaan palvelua käyttää sen lapsikomponenteissa, eli Komponentti 3 ja 4. Angular 2 -sovelluksessa on viisasta ajatella, että jokaisella komponentilla ja moduulilla on oma injektorinsa. Esimerkiksi Komponentti 4:n injektori, on Komponentti 1:en ja moduulin lapsi-injektori, se sisältää komponentin ja moduulin rekisteröimät palvelut. (Google [5] 2017)



Kuva 9. Ohjelmistorakenne

Riippuvuuksien injektointi on isossa osassa ohjelmiston rakennetta ja pitääkin miettiä, missä osassa ohjelmaa halutaan tietyt palvelut rekisteröidä tarjoajiksi. Tahtotilanne voi vaihdella palvelun toteutuksen mukaan, esimerkiksi jaetut palvelut tulisi rekisteröidä tarjoajiksi ohjelman juurimoduulissa mutta yleisesti tilanne riippuu palvelusta. Jos palvelua vaaditaan esimerkiksi vain jossain tietyssä osaa ohjelmaa, se voidaan rekisteröidä sen osan juurikomponentissa tai mahdollisessa ominaiskohtaisessa moduulissa. (Google [5] 2017)

HTML-mallipohjien Angular-ominaisuudet ovat myös kokeneet muutoksia, esimerkiksi Angular 1 -suodattimet ovat nykyään putkia. Useimmat suodattimet ovat tulleet suoraan uuteen versioon putkina mutta joitain eroavaisuuksia löytyy. Esimerkiksi filter ja orderBy on poistettu kokonaan suorituskykysyistä. Toiminnot tulee implementoida suoraan komponentin ohjelmakoodiin. Ilmaisuja koskien suurin ero on siinä, että interpoloinneissa ja sidoksissa ei enää tarvita viittausta komponenttiin. Angular 2:ssa ilmaisu viittaa aina kyseiseen mallipohjaan assosioituun komponenttiin. Angular 2 ei sisällä ollenkaan bootstrap-direktiiviä. Bootstrap-direktiivejä käytettiin Angular 1:ssä sovelluksen käynnistämiseen, mutta parhaana

käytäntönä on kuitenkin pitkään ollut bootstrappaaminen ohjelmakoodissa angularin bootstrap funktiolla. (Google [6] 2017)

Muutokset Angularin sisäänrakennettuihin direktiiveihin ovat pieniä. Monet Angular 1:n direktiivit kirjoitetaan väliviivalla esimerkiksi ng-class, mutta Angular 2:ssa väliviiva on poistettu ja aloitettu seuraava sana yhteen isolla kirjaimella. Tyyli on niin sanottu camelCase-kirjoitustyyli, missä on pieni alkukirjain ja seuraavat sanat on eroteltu isoilla kirjaimilla. Muuttuneista direktiiveistä käytetyimmät löytyvät kuvasta 10. Ng-repeat on Angular 2:ssa *ngFor, edeltävä tähti on pakollinen ja se löytyy kaikista rakennedirektiiveistä. Rakennedirektiivi tarkoittaa HTML-direktiiviä, joka vaikuttaa renderöitävän sivun rakenteeseen. *ngForilla esimerkiksi tulostetaan sivulle listan sisaltö sen koon mukaan, let sanalla määritetään muuttuja eli movie ja listan prepositio on Angular 2:ssa of, ei in. (Google [6] 2017)

*ngIf on myös rakennedirektiivi ja sisältää myös pakollisen tähden, Angularin 1 version ng-if korvattiin tällä. NgIf:n toiminta on kuitenkin täysin vastaava kuin Angular 1 versiossa, eli tässä näytetään taulukko, jos movies kokoelmalla on pituus. Kaksisuuntainen tiedon sitominen vaikutti positiivisesti Angular 1:en suosioon, Angular 2:sta löytyy myös ”two-way binding”, mutta toteutus on erilainen. [(ngModel)] syntaksi on käytännössä vain syntaksisokeria sille, että määritetään sitominen ominaisuudelle ja tapahtumalle, eli eventille ja propertylle. Kaksisuuntaisen siitä tekee sen, että ominaisuus sidotaan Angular 2 -komponentista näkymälle, eli HTML-mallipohjalle ja tapahtuma sidotaan näkymältä komponentille. (Google [6] 2017)

```
<tr *ngFor="let movie of movies">
<table *ngIf="movies.length">
<input [(ngModel)]="favoriteHero" />
```

Kuva 10. HTML-mallipohjadirektiivien esimerkit (Google [6] 2017).

4 TUTKIMUSASETELMA

4.1 Tutkimuksen tarkoitus

Tutkimuksen idea lähtee toimeksiantajayrityksen tarpeesta päivittää käyttöliittymäpuolen Angular 1 -ohjelmistokoodi versioon 2. Yrityksessä on sisäisesti keskusteltu migraation mahdollisuudesta, mutta opinnäytetyöprojektin alkaessa virallisia keskusteluja tai palavereita ei ollut käyty. Tutkimuksen tarkoitus oli selvittää, mitä hyötyjä ja mahdollisia haittoja migraatiosta voisi seurata, miten migraatio onnistuu sovellukseen, miten se tulisi suorittaa ja kuinka työläs migraatio tulisi olemaan. Työn varsinainen tulos on migraatiomenetelmä, jonka avulla yritys voi suorittaa versiopäivityksen. Tiedot migraation tarpeista ja syistä on saatu haastattelemalla yrityksen tuotekehityksen työntekijöitä.

Muutokset Angular 1:en ja Angular 2:en välillä vaihtelevat riippuen Angular 1 -ohjelmakoodin tyylistä. Jos esimerkiksi Angular 1 -versio on toteutettu John Papan parhaiden käytäntöjen mukaan ja kirjoitettu TypeScriptillä, on migraatioprojekti vaivattomampi (Papa 2016). Enemmän kuitenkin migraatioprojektia helpottaa jo teoriaosuudessa mainittu Angularin 1.5 -version mukana tullut component API. Yrityksen sovelluksessa component API:a noudatetaan vain harvoissa komponenteissa, ja suurin osa sovelluksen komponenteista on kirjoitettu direktiiveiksi tai komponenttidirektiiveiksi.

Toimeksiantajayrityksen mielestä migraatio Angular 2:een tuo varmuutta jatkuvalle tuotteelle. Kirjoitushetkellä Angular 1:llä pystytään toteuttamaan kaikki yrityksen tarpeet, mutta sovellukselle on suunniteltu pitkä elinkaari, joten tulevaisuudessa saattaa tulla ominaisuuksia, joita ei enää Angular 1:llä pystytä toteuttamaan. Web-ohjelmointi on myös kehittynyt valtavasti Angular 1:en julkaisemisen jälkeen, eikä kaikkia sen ongelmia pystytä korjaamaan pienillä versiopäivityksillä. Haastattelemani henkilöt olivat vahvasti sitä mieltä, että Angular 2 tuo lisää tuottavuutta ja laatua ohjelmistokehitykseen. Oma näkemykseni on, että ainakin henkilöille joilla ei ole paljoa kokemusta Angular-kehittämisestä, on Angular 2 selvästi helpompi oppia. Suorituskykyparannukset ovat myös yritykselle kiinnostava aihe, mutta ei ole mitään

varmuutta, että parannuksilla olisi mitään vaikutusta yrityksen sovelluksen suorituskykyyn.

Toimeksiantajayritys näki hybridiapplikaation parhaana migraatiovaihtoehtona. Haastattelujeni mukaan kokonainen ja suora uudelleenkirjoittaminen olisi liian työläs, ja veisi todennäköisesti resursseja erilaisista toiminnallisista muutoksista. Hybridiapplikaatiolla on hyvät puolensa, mutta toteutus tulee olemaan monimutkainen yrityksen suurikokoisessa sovelluksessa. Sovelluksen komponentit ovat riippuvaisia toisistaan ja sovelluksessa käytetään paljon koko sovelluksen kattavia palveluja. Riippuvuuksien injektoinneissa pitää olla varovainen, ja sopia tyyli sille, että missä riippuvuuksia halutaan injektoida.

4.2 Tutkimusmenetelmä

Ensimmäiseksi suoritettiin testisovelluksena tutkimuksessa käytettävän ajanvarausjärjestelmän migraatio, missä oli tarkoitus saada alustava näkemys Angular 2 kehittämisestä. Testisovelluksen migraatio suoritettiin uudelleenkirjoittamalla koko sovellus Angular 2 -versiolla. Testisovellus on kehitetty Satakunnan Ammattikorkeakoulun Escape Roomin ajanvarausjärjestelmäksi. Ajanvarausjärjestelmässä on kalenteri ja kaavio. Kalenterista valitaan päivä, jolloin kaaviossa näytetään, onko kyseiselle päivälle vapaita varauksia. Jos päivälle on vapaita aikoja, tulostetaan vapaat ajat aikavalinnalle tehtyyn valintaelementtiin. Kun aika on valittu, täytetään tarvittavat yhteystiedot ja lähetetään varaus. Varaus käsitellään serveripuolen ohjelmistokoodissa ja syötetään tietokantaan.

Lähtötilanne testisovelluksessa oli Angularin dokumentaation listaamia käytäntöjä noudattava ja TypeScriptillä toteutettu Angular 1 applikaatio. Sovelluksessa on ydinkontrolleri ja kaksi direktiiviä: kalenteri ja kaavio. Data direktiivien ja kontrollerin välillä liikkuu Angularin scope:n avulla. Kalenterista valittu päivä lähetetään ydinkontrolleriin, ydinkontrollerissa haetaan tietokannasta kaikki varattavat ajat ja suodatetaan ne valitun päivän mukaan ja lähetään kaavion direktiiviin. Kaavion direktiivissä tulostetaan vapaat ajat kaaviolle, lähetetään käyttäjän syöttämä data palvelinsovellukselle ja valitun ajan tila muutetaan varatuksi.

Yrityksen ohjelmisto ja ajanvarausjärjestelmä noudattavat samaa Angular 1 -tyyliopasta, joten ajatuksena oli, että testiohjelmiston migraatiota pitäisi pystyä hyödyntämään ainakin joitain komponentteja konvertoidessa, kun yrityksen Angular 2 -migraatio aloitetaan. Valmiit migraatiomenetelmät tulevat kuitenkin olemaan hyvin erilaiset.

Molemmat sovellukset käyttävät TypeScriptiä, eikä tutkimuksessani oteta kantaa JavaScriptin muuttamisesta TypeScriptiin. TypeScript -migraatio ei tule olemaan osana migraatiomenetelmääni. Oli myös otettava huomioon, että Angular 2:sta ei ollut vielä julkaistu virallista versiota tutkimusprojektia aloitettaessa. Angular 2 oli julkaisukandidaattivaiheessa ja muutoksia oli odotettavissa.

5 TUTKIMUS

5.1 Testisovelluksen Angular 2 -migraatio

Migraatioita aloittaessa oli otettava huomioon, että Angular 2:n arkkitehtuuri on erilainen. Angular 1 -sovellukset rakennettiin kontrollereista, direktiiveistä ja moduuleista, kun taas Angular 2 käytettäessä sovellus rakennetaan pääosin komponenteilla ja moduuleilla. Ohjelmiston ydin on Angular 2:ssa moduuli (ngModule), missä tuodaan tarvittavat moduulien muodossa olevat Angularin valmiit kirjastot sovellukseen. Moodulissa asetetaan myös sen tarvitsemat komponentit ja palvelut, sekä esiladataan juurikomponentit. Erillisessä tiedostossa esiladataan taas moduulit.

Testisovelluksen Angular 2.0 -versiossa on juurikomponentti, jossa asetetaan sovelluksen ensivalitsija ja HTML-pohja. Sovellus rakentuu kahdesta komponentista: Angular 1.5 -versiossa olleet kalenteri ja kaavio-direktiivit on muutettu komponenteiksi. Komponenttien sisäistä toiminnallisuutta ei tarvinnut muuttaa, mutta komponenttien vuorovaikutus toteutettiin jaetulla palvelulla, joka esiladataan sovelluksen ydinkomponentissa ja sitä pystytään käyttämään kaikissa sen alikomponenteissa. HTTP-kutsuja ei suositella suorittamaan suoraan komponenteissa, vaan ne tulisi aina toteuttaa palvelujen avulla (Google [9] 2017). Tästä syystä sovellukseen on tehty kaksi erillistä palvelua. Toinen on tarkoitettu kaavioon täytettyjen tietojen palvelinsovellukselle lähettämistä varten, toinen jaettua dataa ja vapaiden varauksien hakemista varten.

Ongelmakohtien ratkaisuun tehtiin tutkimustyötä ja arvioitiin, mikä mahdollisista toimintatavoista olisi tilanteessa paras. Toimintatapavaihtoehtoja oli tilanteessa kaksi: sovelluksen kokonainen uudelleenkirjoittaminen tai hybridiapplikaatio. Uudelleenkirjoittamisessa migraation voi aloittaa tekemällä Angular 2 -sovellus alusta loppuun samanaikaisesti, kun Angular 1 -versio on vielä käytössä. Hybridiapplikaatiovaihtoehto tarkoittaa sitä, että sovellus sisältää samanaikaisesti Angular 1 ja 2 -koodia. Näin päivitys suoritetaan komponentti kerrallaan käyttäen

Angular 2:n omaa UpgradeAdapter moduulia, jolla päivitetään Angular 1 -komponentteja tai päivitetään alaspäin Angular 2 -komponentteja (Precht 2016).

Projektin koko vaikuttaa toimintatapojen olennaisuuteen. Esimerkiksi suuressa projektissa koko ajan päivitettävä Angular 1 -versio ja samanaikaisesti alusta loppuun rakennettava Angular 2 -versio vaatisivat paljon resursseja. Suureen projektiin UpgradeAdapterin käyttäminen sopii hyvin, mutta komponenttien riippuvuuksien kanssa voi tulla ongelmia, jos esimerkiksi Angular 2 -komponenttien tarvitsemat Angular 1 -palvelut eivät ole päivitettävissä. Pienessä projektissa, joka sisältää esimerkiksi vain 1-10 komponenttia, voi sovelluksen uudelleenkirjoittaminen olla järkevämpi vaihtoehto. Ajanvarausjärjestelmän migraatioon valittiin suoraan uuden version tekemisen Angular 2:lla pääasiassa komponenttien määrän ja ennustetun työmäärän takia.

Komponenttien välisestä interaktiosta on useita blogikirjoituksia, ja se onkin yksi puhutuimpia aiheita Angular 2:een siirtymisessä. Angular 1:ssä käytettiin pääasiassa kaksisuuntaista datan sitomista, eli dataa voitiin siirtää komponentilta komponentille molempiin suuntiin. Kumpikin komponentti pystyy tässä tilanteessa siis muokkaamaan siirrettyä dataa ja muutos välittyy toiselle komponentille. Tätä samaa tyyliä ei enää versioon 2 haluttu tuoda, vaan uudessa versiossa käytetään yksisuuntaista datan virtausta. Komponentti Y voi sisältää esimerkiksi muuttujan, jonka sisältöä käyttäjä muokkaa Web-sovelluksen lomakkeella. Käyttäjän hyväksyessä lomakkeen painamalla nappulaa, lähetetään muuttuja komponentille Y. Komponentti Y näin voi käyttää muuttujaa. Operaatiota kutsutaan datan siirtämiseksi ”Parentin” ja ”Child” -komponentin välillä (Motto 2016 [1]). Tähän käytetään ”Input” ja ”Output” -dekoraattoreita. Input:ia käytetään datan siirtämiseen tilanteissa, joissa siirrettyä dataa ei tarvitse erikseen kuunnella. Jos komponentin halutaan saavan tiedon muutoksista, käytetään Output-dekoraattoria ja EventEmitter:iä (Motto 2016 [2]).

5.2 Angular 2 -migraatio yrityksen sovellukseen

Ensimmäinen askel migraatiossa oli Angular 2:n asentaminen, jonka jälkeen tarkoituksena oli luoda yksinkertainen Angular 2 -komponentti ja saada se näkymään yrityksen sovelluksessa. Operaatio oli tarkoitus toteuttaa UpgradeAdapter -luokkakirjaston sisältämällä downgradeNg2Component metodilla. Idea on, että Angular 1 -sovellukseen voidaan luoda Angular 2 -komponentti ja sitä voidaan käyttää alennettuna Angular 1 -komponenttina (Google [8] 2017). Kyseinen operaatio kuulostaa yksinkertaiselta, mutta suuressa projektissa sen voi tehdä työlääksi käännösympäristön konfiguraatio ja komponenttien riippuvuudet (Caetano 2016).

Käännösympäristön konfigurointi vaati huomattavasti enemmän aikaa verrattuna testisovellukseen. Alkuun ongelmia aiheuttivat TypeScriptin konfiguraatio, riippuvuudet ja kolmannen osapuolen kirjastot. Testiprojektina toteutetun ajanvarausjärjestelmän migraatiossa moduulien tuomisen asetus oli oletuksena jo sama, kuin Angular 2:n kanssa käytettäväksi suositeltu ”moduleResolution: node”. Yrityksen käännösjärjestelmässä oli käytössä classic asetus moduleResolutionille. Module Resolution on se prosessi, jota TypeScriptin kääntäjä käyttää ymmärtääkseen mitä tuotu moduuli referoi (Microsoft 2016). Asetuksen muuttaminen tarkoitti yrityksen omien moduulien tuontipolkujen korjaamista.

Sovelluksen riippuvuussuhteet vaikuttavat migraatioprojektin suunnitteluun. Käyttämällä UpgradeAdapter luokkakirjaston metodia upgradeNg1Provider voidaan päivittää Angular 1 -palvelut Angular 2 -palveluiksi. Ongelmaksi yrityksen tilanteessa tulee palvelujen injektiot eri komponenteissa. Ovatko injektoidut palvelut sillä tasolla, että niitä voidaan päivittää Angular 2 -palveluiksi ja pystytäänkö päivitettyä palvelua käyttämään jaetusti niin, että data liikkuu Angular 1- ja 2 -komponenttien välillä (Google [2] 2017)?

Komponentit ja luokat, jotka eivät suoranaisesti ole Angular-komponentteja tai -palveluja, mutta käyttävät Angular 1 -spesifisiä tyyppejä ovat migraatiotilanteessa hankalia. Tälläisiä luokkia ei voida suoraan päivittää Angular 2:n UpgradeAdapterilla, eikä muuttaa tyypityksiä Angular 2 -spesifisiksi. Nämä luokat välittävät dataa

ohjelmiston komponenttien välillä, joten luokan tulee toimia hybridisovelluksessa molemmilla Angularin versioilla. Vaihtoehtoina on muokata luokkien toteutusta niin, että niissä ei ole mitään Angular-spesifisiä muuttujia, tai tehdä molemmille versioille omat luokkansa. Luokkien toteutuksen muokkaaminen on parempi vaihtoehto. Rinnakkaiset luokat eri ohjelmistokehyksen versioille kuulostaa ja saattaisi olla operaationa nopeampi, mutta luokat tulisivat olemaan kopioita toisistaan muutamilla muutoksilla.

Muutoksia komponentteihin aiheuttaa kolmannen osapuolen tai jopa Angularin omien kirjastojen muutokset Angular 2:een verrattuna, esimerkiksi Angularin 1:n angular-ui-bootstrapin puuttuminen Angularin 2 versiossa. UI-bootstrap on Angularin ratkaisu käyttöliittymän ulkonäköön ja käytettävyyteen ja sitä käytetään laajalti yrityksen komponenteissa. UI-bootstrapin Angular 2 -versio on vielä työn alla, mutta se käyttää Bootstrap-ohjelmistokehyksen alpha-vaiheessa olevaa versiota 4. Yrityksen sovelluksessa on käytössä virallisesti julkaistu ja angular-ui-bootstrapissa suositeltu Bootstrap 3.3.6. Vaihto uudempaan versioon ei onnistu, koska vanhat tyylitiedostot eivät toimi uudemmalla versiolla. Paremmaksi vaihtoehdoksi yrityksen tilanteeseen sopii Valor Softwaren ng2-bootstrap, joka on yhteensopiva molempien aiemmin listattujen Bootstrapin versioiden kanssa ja erityisesti tärkeänä yritykselle on yhteensopivuus Bootstrap 3.3.6 version kanssa.

Sovelluksessa käytetään erityisesti modaaleja Angular UI-bootstrap tyylillä, jossa modaali-ikkunan malli luodaan jossain komponentissa, mutta modaali-ikkunaa käytetään tietyn toiminnon yhteydessä toisessa komponentissa. Ng2-bootstrapin toteutus operaatiolle on parhaimmillaankin kömpelö. Modaali-ikkunoita varten suosittelenkin valitsemaan angular2-modal komponentin, joka on tehty vain modaalien implementointiin Angular 2 -sovelluksessa. Se sopii hyvin yrityksen käyttötarkoitukseen.

Angular 1 -sovelluksessa käytössä ollut $timeout eli TimeoutService on myös poistettu kokonaan Angular 2 -versiosta. JavaScriptin ajastusoperaatiot setTimeout() ja setInterval() vievät tarkoituksen $timeout:lta. Yrityksen käyttämä angular-translate

on implementoitu lähes sellaisenaan myös Angular 2:en ng2-translate nimellä. Ng2-translate sisältää jopa samat metodit ja metodeille annettavat parametrit.

Moduulien tuominen ja niputtaminen aiheuttavat ongelmia monissa ohjelmistoprojekteissa, vaihtoehtoja toimintojen suorittamiseen on useita. Angular suosittelee ohjeissaan käyttämään SystemJS:ää tai Webpackia (Angular Webpack 2016). Yrityksen sovelluksessa on käytetty Almondia ja Gulpia. Gulpilla suoritetaan moduulien niputtaminen ja Almondilla AMD-moduulien lataaminen. Angular 2 -kirjastojen lataaminen koitui jo ongelmalliseksi, eikä se onnistunut alun perin AMD-moduuleilla. Väliin tarvittiin komponentti, jolla käytiin läpi Angularin omat moduulit ja palautettiin ne muodossa, jossa Almond osaa niitä käsitellä. Tämä komponentti kuitenkin on vain väliaikainen apu ongelmaan, joka voi toistua kolmannen osapuolen moduuleja tuotaessa yrityksen projektiin. Aiemmin mainittu Ng2-bootstrap aiheutti jo nyt vaikeuksia, se ei ole suoraan yhteensopiva Almondin ja AMD-moduulien kanssa. Muutamien epäonnistuneiden yritysten jälkeen asiasta neuvoteltiin yrityksen työntekijöiden kanssa ja tehtiin päätös lähteä tutkimaan SystemJS:ää ja Webpackia moduulilataajana ja niputtajana.

Päätettiin aloittaa tutkimus Webpackin käyttöönotosta, koska Webpack tulisi todennäköisesti olemaan paras vaihtoehto hybridiapplikaation pohjana mm. Angularin kehitystiimin tuen ja Angular-CLI -yhteensopivuuden takia. Angularin dokumentaatio tarjoaa hyvät ohjeet Webpackin konfigurointiin kehitys- ja tuotantoympäristössä (Google [7] 2017). Testiin otettiin Webpackin uusin virallinen versio. Alun ongelmien jälkeen huomattiin, että uudet TypeScript 2.0:n konfiguraatioasetukset eivät toimi virallisten Webpack 1 -versioiden kanssa. Yksi näistä uusista asetuksista on baseUrl, joka yrityksen sovelluksessa on asetettu. Sillä säädetään ympäristön juuriosoite, ja moduuleja voidaan tuoda komponentteihin suhteessa tähän polkuun. Webpackin versio 2 mahdollistaa uusien TypeScript 2:en asetusten käyttämisen ympäristössä.

Yrityksen sovellukseen tehtiin alustavan Webpack konfiguraation, jossa niputetaan TypeScript-tiedostojen sisältö yhteen tiedostoon. Tähän tiedostoon viitataan sovellusta avattaessa. Webpack jätetään taustalle tarkkailemaan TypeScript- ja HTML-tiedostoja, ja muutoksia tehtäessä suoritetaan niputtaminen uudelleen. Ensimmäinen niputuskerta kestää noin 30 sekuntia, mutta tarkkaillessa muutoksien niputukseen

menee vain noin 3 sekuntia. Niputtajan suorituskyky on tärkeä erityisesti yrityksille ja suurille ohjelmistoprojekteille. Jos muutoksien jälkeiseen niputukseen menisi esimerkiksi 30 sekuntia, päästäisiin muutoksia testaamaan vasta sen jälkeen. Niputtajan hitaus vaikuttaa siis suoraan työntekijän tehokkuuteen.

Yrityksellä on myös käytössä oma moduuli, joka tuo sovellukseen tyylikirjaston ja käyttöliittymäkomponentteja. Moduuli on tehty toimivaksi AMD-moduulina ja erityisesti tuotavaksi Almond-lataajaa käyttäen. Tyylit tuotiin Angular 2 komponenttien käyttöön väliaikaisella ratkaisulla, mutta kirjaston direktiivejä ei voi käyttää missään ohjelman Angular 2 osassa. Esimerkiksi yrityksen latausikonit ja jotkin nappulat eivät näy Angular 2:lla toteutetuissa sovelluksen osissa. Yrityksen pitääkin tehdä moduulista sopiva molemmille Angularin versioille tai vaihtoehtoisesti tehdä uusi pelkästään Angular 2:lle.

Migraation ohjelmointiosion oletettiin olevan suoraviivainen, koska yrityslogiikkaa ei Angular-migraation yhteydessä tarvitse muuttaa. Tehtävä on parhaassa tapauksessa vain komponenttidirektiivien päivittämistä Angular 2 -muotoon tai komponenttien ja direktiivien sulauttamista yhteen. Tämä on siis tilanne, jossa John Papan Angular Style Guidea noudatetaan koko sovelluksessa (Papa 2016). Tilanne oli kuitenkin yrityksen sovelluksessa toinen. Sovellus sisältää tyyliopasta noudattavia komponentteja, mutta myös vanhempaa tyyliä olevia komponentteja. Haastavaa migraatiosta tekeekin juuri ohjelmiston monimuotoisuus. Kontrollerien ja direktiivien toteutukset vaihtelevat toteutusajankohdasta riippuen. Uusimmat komponentit muistuttavat tai ovat John Papan ja suositun Angularin oman tyylioppaan mukaisia, mutta vanhemmat ovat vielä aikaisempien Angular versioiden tyylisiä. Todd Motto kertoo blogissaan, kuinka uusi Angular 1.5 component() metodi helpottaa ohjelmoijia käyttämään parhaita käytäntöjä ja tekemään Angular 1 koodista Angular 2:ta muistuttavaa koodia (Motto 2016 [3]). Direktiivien ja komponenttidirektiivien muokkaus Angular 1 -komponenteiksi helpottaa migraatiota Angular 2:en, mutta eri asia on, onko se järkevää tässä yrityksen tilanteessa. Ratkaisu kannattaa tehdä yritysten resurssien ja osaamisen mukaan.

6 TUTKIMUSTULOS

Pienen tai yksinkertaisen Angular 1 -sovelluksen migraatio Angular 2 -sovellukseksi ei vaadi paljon työtä kehittäjältä, jolla on kokemusta Angular 1 -sovelluksista. Jos kokemus on Angular 1.5 component API:a noudattavasta kehittämisestä, pitäisi migraation sujua sulavasti. Suurimmat muutokset ovat todennäköisesti komponenttien välisessä interaktiossa, missä dataa liikutetaan komponentilta toiselle. Tämän suorittamiseen suositellaan Angular 2 sovelluksessa jaettua palvelua ja Rxjs:n Observablejen hyödyntämistä. Palvelu sisältää Observable-luokan lähteen ja virran. Dataa muokkaavassa komponentissa käytetään kuvan 11 tapaan palvelun lähdettä ja muutoksia kuunnellaan kuvan 12 mukaan toisessa komponentissa käyttämällä Rxjs Subscribtion:ia ja virran subscribe-funktiota.

```
private selectedDay = new Subject<moment.Moment>();

selected$ = this.selectedDay.asObservable();

constructor (private http: Http) {
    this.dateSelected = moment(this.selectedDay.next);
    this.http.get('http://localhost:8080/api/reservations')
        .toPromise()
        .then((data => this.reservationContainers = data.json()))
        .catch((data) => alert('Error connecting to server'));
    this.getReservationContainer(this.dateSelected);
}
```

Kuva 11. Määritetään dataService palvelussa Moment tyyppinen Subject ja asetetaan se lähteeksi

```
constructor (private dataService: DataService) {
    this.subscription = dataService.selected$.subscribe(
        result => {
            this.calendarDate = result;
    });
}
```

Kuva 12. Kuunnellaan dataService -palvelun lähdettä ja asetetaan tulos muuttujaan

Suuri osa komponenttien interaktiota käsittelevistä blogikirjoituksista on Input- ja Output-dekoraattoreista. Tämä voi johtua vain siitä, että Input- ja Output-dekoraattorien käyttö on uusi ominaisuus Angular 2:ssa. Aiemmin mainittu jaettu

palvelu on ollut käytössä jo Angular 1 -sovelluksissa. Henkilökohtaisesti en näe syytä olla käyttämättä jaettua palvelua. Tilanteessa, jossa komponenttien välillä liikkuu vain muutaman muuttujan sisältämä data, on Input- tai Output-dekoraattorien käyttö viisaampaa. Itse näen valinnan kuitenkin mielipidekysymyksenä.

Ongelmat moduulilataajista ja niputtajista veivät valtavasti aikaa. Tuloksena kuitenkin saatiin hybridiapplikaatiopohja kahdelle eri moduulilataajalle, Webpackille ja Almondille. Havaitut ongelmat Almondin kanssa saavat minut suosittelemaan Webpackin käyttöönottoa yritykselle ja muille samassa tilanteessa migraatiota suunnitteleville. Webpackiin vaihtaminen saattaa viedä aikaa, mutta sen käyttöönottamisella vältytään tulevaisuudessa mahdollisilta ongelmilta kolmannen osapuolen komponenttien, moduulien ja kirjastojen kanssa. On myös mahdollista, että tulevaisuudessa Almondin Angular 2 -tuki paranee. Se vaatii joko Almondin tai Angularin moduulien muokkaamista, mutta tällä hetkellä ongelmat Almondin kanssa alkavat jo Angular 2:den tärkeimmistä ydinmoduuleista, joita tarvitaan jokaisen Angular 2 -sovelluksen kehittämiseen.

Toimeksiantajayrityksen sovelluksesta saatiin konvertoitua tietty eristetty osa toimimaan Angularin 2 -versiolla. Eristetty osa koostuu kahdesta komponentista: tuotelistasta ja tuotetiedoista. Kun tämä kyseinen osa sivusta avataan, tulostuu näytölle tuotelista, kun taas tuotelistassa olevaa tuotetta klikataan, mennään kyseisen tuotteen tuotetietoihin. Vaikeuksia osan migraatioon toi Angular 1 -sovelluksessa käytetty moduulien konfiguroinnin puuttuminen Angular 2:sta, joten ohjelmistokehysten komponenttien väliin tarvitaan ylimääräinen Angular 1 -komponentti, jossa komponentille konfiguroidaan reitti module.config() funktiolla. Angular 1 -komponenttiin injektoidaan myös tarvittavaa dataa palveluilta, joita ei pysty muokkaamatta päivittämään Angular 2:en UpgradeAdapterilla. Injektoitu data lähetetään eteenpäin Angular 1 -komponentilta Angular 2 -komponentille käyttämällä Angular 2:en @Input dekoraattoria. Angular 1 -komponentin HTML-mallipohjassa instantioidaan Angular 2 -komponentti. Angular 2 -komponentti sisältää logiikan ja sivun HTML-mallipohjan.

Komponenttien sisältämä logiikkaa ei tarvinnut suuria muutoksia. Sovelluslogiikka ei sisältänyt mitään Angular 1 -spesifisiä ominaisuuksia. Tarkoituksena olikin saada

näkymään vain tarvittava data tulostettuna Angular 2 -komponentilta hybridiapplikaatiolle. Tavoitteessa onnistuttiin: tuotelista ja tuotetiedot ovat nyt Angular 2 -komponentteja Angular 1 -välikappaleilla. Suoritettu migraatio tähän osaan on kuitenkin vain alustava, esimerkiksi sovelluksessa käytettyjä Angular 1 -käyttöliittymäkomponentteja ei voida käyttää Angular 2 -komponenteissa, eli Angular 2:lla toimiva osa ei sisällä tiettyjä käyttäjää helpottavia nappeja ja latausikoneja.

Ennen migraatioprojektin kokonaista aloittamista tuleekin yrityksellä olla käyttöliittymän tyylikirjastosta tehtynä Angular 2 -versio. Angular 2:n UI-komponenttikirjastot ovat tällä hetkellä keskeneräisiä, joten tyylikirjaston konvertoimisella ei kannata kiirehtiä. Yritys voi sen sijaan aloittaa heti uuden koodin tuottamisen Angular 1.5 -version component API:a noudattaen, jolla Angular 1 -komponentit saadaan muistuttamaan Angular 2 komponentteja ja näin tulevaa migraatiotyötä helpotetaan. Vanhoja direktiivejä tulisi myös konvertoida 1.5 -komponenteiksi, kun niihin tehdään uusia ominaisuuksia. Yrityksen tulee myös seurata tulevia Angular 1:n uusia ominaisuuksia. Google on ilmoittanut, että uudet ominaisuudet tulevat olemaan Angular 2 -yhteensopivuuteen viittaavia ja migraatiotyötä helpottavia.

Migraatioprojektin tyyliin on kaksi vaihtoehtoa, hybridisovellus ja sovelluksen kokonaan uudelleenkirjoittaminen Angular 2:lla. Hybridisovellus sopii hyvin suuriin sovelluksiin, mutta hybridisovellus kasvattaa kuitenkin tiedostokooltaan niputettua tuotantopakettia ja sovelluksen ensilataus hidastuu. Angular 2:n lisääminen kasvattaa kirjastopaketin kokoa toimeksiantajayrityksen sovelluksessa kaksinkertaiseksi. Ero latausnopeudessa esimerkiksi mobiililaitteella voi olla huomattava. Ohjelmakoodia ajatellen hybridisovellus on moduuleiltaan monimutkainen. Angular 2 -moduulit sisältävät päivitettyjä Angular 1 -komponentteja ja -palveluita tai alaspäin päivitettyjä Angular 2 -komponentteja ja -palveluita. Komponenttien riippuvuuksia joudutaan päivittämään kehykseltä toiselle aina, kun riippuvuudet ovat toteutettu komponenttiin nähden eri versiolla. Tutkimukseni mukaan kehitettyjä hybridisovelluksia on vähän. Yritykset ja yleisesti ohjelmoijat ovat ensisijaisesti valinneet uudelleenkirjoittamisen migraatiotyylikseen. Uudelleenkirjoittaminen voi ollakin parempi vaihtoehto erityisesti sovelluksessa, jossa komponenttien välisiä riippuvuuksia on paljon.

LÄHTEET

Angular University. 2016. AngularJs vs Angular 2 - An In-Depth Comparison. Viitattu 13.12.2016. http://blog.angular-university.io/introduction-to-angular2-the-main-goals/

Caetano, D. 2016. Upgrading Your Application to Angular 2 with ng-upgrade. Viitattu 5.11.2016. http://blog.rangle.io/upgrade-your-application-to-angular-2-with-ng-upgrade/

Cooper, J. 2013. AngularJS Step-by-Step. Viitattu 6.11.2016. https://www.pluralsight.com/blog/tutorials/angularjs-step-by-step-controllers

Google (1). 2010-2016. Angular 1 Developer Guide. Viitattu 10.12.2016. https://docs.angularjs.org/guide/introduction

Google (2). 2010-2016. Making Angular 1 Dependencies Injectable to Angular 2. Viitattu 10.12.2017. https://angular.io/docs/ts/latest/guide/upgrade.html#!#making-angular-1-dependencies-injectable-to-angular-2

Google (3). 2010-2016. Developer Guide: Components. Viitattu 10.12.2016. https://docs.angularjs.org/guide/component

Google (4). 2015-2017. Angular 2 modules, ngModules. Viitattu 12.12.2016. https://angular.io/docs/ts/latest/guide/ngmodule.html

Google (5). 2015-2017. Angular 2 Dependency Injection. Viitattu 12.12.2016. https://angular.io/docs/ts/latest/cookbook/dependency-injection.html

Google (6). 2015-2017. Angular 1 to 2 Quick Reference. Viitattu 12.12.2016. https://angular.io/docs/ts/latest/cookbook/a1-a2-quick-reference.html

Google (7). 2015-2017. Introduction to Webpack. Viitattu 5.1.2017. https://angular.io/docs/ts/latest/guide/webpack.html

Google (8). 2015-2017. Angular 1 to Angular 2 Upgrade Strategy. Viitattu 11.12.2016. https://docs.google.com/document/d/1xvBZoFuNq9hsgRhPPZOJC-Z48AHEbIBPlOCBTSD8m0Y/edit#

Google (9). 2015-2017. HTTP Client. Viitattu 5.1.2017. https://angular.io/docs/ts/latest/guide/server-communication.html

Hamilton, N. 2008. The A-Z of Programming Languages: JavaScript. Viitattu 8.11.2016. http://www.computerworld.com.au/article/255293/-z_programming_languages_javascript

Ivanovs, A. 2016. Top 23 Best Free JavaScript Frameworks for Web Developers 2016. Viitattu 20.11.2016. https://colorlib.com/wp/javascript-frameworks/

Kasireddy, P. 2016. JavaScript Modules: A Beginners Guide. Viitattu 20.11.2016. https://medium.freecodecamp.com/javascript-modules-a-beginner-s-guide-783f7d7a5fcc#.7gyx8mu8c

Lehdonvirta P. & Korpela J. 2013. HTML5 Sovellusalustana. Helsinki: RPS Markkinointi

Microsoft. 2010-2016. Module Resolution. Viitattu 10.1.2017. https://www.typescriptlang.org/docs/handbook/module-resolution.html

Motto, T (1). 2016. Passing Data Into Angular 2 components with “Input”. Viitattu 6.12.2016. https://toddmotto.com/passing-data-angular-2-components-input

Motto, T (2). 2016. Component events with EventEmitter and "Output" in Angular 2. Viitattu 6.12.2016. https://toddmotto.com/component-events-event-emitter-output-angular-2

Motto, T (3). 2015. Exploring the Angular 1.5 .component() method. Viitattu 6.12.2016. https://toddmotto.com/exploring-the-angular-1-5-component-method/

Osmani, A. 2012. Writing Modular JavaScript With AMD, CommonJS & ES Harmony. Viitattu 9.9.2016. https://addyosmani.com/writing-modular-js/

Panda, S. 2014. A Practical Guide to AngularJS. Viitattu 8.9.2016. Directives https://www.sitepoint.com/practical-guide-angularjs-directives/

Papa, J. Päivitetty 2016. Angular 1 Style Guide. Viitattu 23.11.2016. https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#single-responsibility

Parviainen, T. 2015. Refactoring Angular apps to Comnponent style. Viitattu 17.9.2016. https://teropa.info/blog/2015/10/18/refactoring-angular-apps-to-components.html

Precht, P. 2016. Upgrading apps to Angular 2 using ngUpgrade. Viitattu 6.1.2017. http://blog.thoughtram.io/angular/2015/10/24/upgrading-apps-to-angular-2-using-ngupgrade.html

Savkin, V. 2016. Angular2: Why TypeScript? Angular Developer. Viitattu 10.1.2017. https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8#.486xxgxc6