Antti Kettunen

YKSIKKÖTESTAUS ANGULARJS-SOVELLUKSESSA

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma

Toukokuu 2016

MAMK
University of Applied Sciences

KUVAILULEHTI

MAMK University of Applied Sciences	**Opinnäytetyön päivämäärä** 26.5.2016
Tekijä(t) Antti Kettunen	**Koulutusohjelma ja suuntautuminen** Tietojenkäsittelyn koulutusohjelma
Nimeke Yksikkötestaus AngularJS-sovelluksessa	

Tiivistelmä

Sovelluskehitys käy kiivaammin kuin koskaan ennen. Web-teknologioiden kehittyessä yhä useampi palvelu laajentaa toimintaansa Internetiin, ja tämän siirtymän helpottamiseksi luodaan vuosi toisensa jälkeen uusia työkaluja ja sovelluskehyksiä. Uusiin mahdollisuuksiin mukautuminen ei kuitenkaan usein tule ilman hintaa, sillä rikkoutumisen riski olemassa olevissa sovelluksissa on suuri, kun niihin tehdään muutoksia.

Tätä riskiä on kuitenkin mahdollista minimoida järjestelmällisen testauksen kautta. Opinnäytetyöni käsittelee yhtä tällaista testauksen muotoa - ohjelmistotestausta - joka tarkastelee ohjelman testaamista teknisestä näkökulmasta. Keskityn työssä erityisesti yksikkötestaukseen, jonka tarkoituksena on varmistaa ohjelman pienimpien osien oikeanlainen toimivuus.

Toimeksiantona tutkin, kuinka yksikkötestausta on mahdollista suorittaa sovelluksessa, joka on kehitetty AngularJS-sovelluskehyksellä. Lopputuloksena syntyi käyttövalmis testausympäristö, sekä joukko erilaisia testitapauksia, jotka demonstroivat kuinka testaus tapahtuu käytännössä. Työ toteutettiin olemassa olevaan Metsäpaikka-sovellukseen, jonka kehittämisestä vastaa Mikkelissä toimiva yritys MHG Systems Oy.

Asiasanat (avainsanat)

AngularJS, JavaScript, testaus, ohjelmistotestaus, yksikkötestaus

Sivumäärä 33	**Kieli** Suomi
Huomautus (huomautukset liitteistä)	
Ohjaavan opettajan nimi Arto Väätäinen	**Opinnäytetyön toimeksiantaja** MHG Systems Oy

DESCRIPTION

MAMK University of Applied Sciences	**Date of the bachelor's thesis** 26 May 2016
Author(s) Antti Kettunen	**Degree programme and option** Business Information Technology
Name of the bachelor's thesis Unit testing an AngularJS application	

Abstract

The software industry is growing faster than ever. As web technologies improve, more and more companies are starting to either expand or fully move their services to Internet-based solutions. New tools and frameworks are being developed constantly to make this transition a bit easier, but adapting an existing application to these tools often comes with a price. As developers move from one technology to another, the risk of something breaking is high.

This risk can, however, be minimized through various, well-thought-out testing procedures. In this thesis I covered software testing, which approaches testing from a technical standpoint. The main focus of the work was on unit testing, which attempts to ensure that even the smallest pieces of an application's code are functioning correctly.

The assignment for this thesis was to find out how to unit test an application built with the JavaScript framework AngularJS. The result of this was a fully functional unit testing environment, accompanied with various test cases for demonstrating how unit testing is done in practice. The testing tools were implemented to an existing application called Woodlandmanager, developed by a company in Mikkeli called MHG Systems Ltd.

Subject headings, (keywords) AngularJS, JavaScript, testing, software testing, unit testing	
Pages 33	**Language** Finnish
Remarks, notes on appendices	
Tutor Arto Väätäinen	**Bachelor's thesis assigned by** MHG Systems Ltd

SISÄLTÖ

1 JOHDANTO

Elämme aikaa, jossa teknologian merkitys ihmisen arjessa korostuu päivä päivältä. Internet ja sitä hyödyntävät laitteet jatkavat yleistymistään, minkä seurauksesta perinteisetkin palvelut voidaan yhä useammin löytää myös sähköisessä muodossa. Tällainen äkillinen kysynnän ja tarjonnan nousu sovellusmarkkinoilla on pakottanut sovelluskehitystä kasvamaan nopeammin kuin koskaan ennen, mikä on tuonut uusien mahdollisuuksien lisäksi mukanaan myös haasteita.

Pysyäkseen jatkuvasti kehittyvien tekniikoiden ja muoti-ilmiöiden vauhdissa, kehittäjien tulee kyetä mukautumaan muutoksiin nopeasti. Tämä ei kuitenkaan aina ole yksinkertaista, sillä muutokset tehdään useimmiten olemassa olevan sovelluksen päälle, jolloin vanhan toiminnallisuuden rikkoutumisen riski on suuri. Tämän välttämiseksi kehitysprosessin tueksi voidaan laatia joukko erilaisia testitapauksia, joiden kautta muutosten teko ja niiden vaikutusten seuraaminen helpottuu.

Käsittelen opinnäytetyössäni yhtä tällaista testauksen muotoa - ohjelmistotestausta - joka keskittyy sovelluksen testaamiseen teknisestä näkökulmasta. Käyn läpi eri tasoja, joilla sitä on mahdollista suorittaa sekä sitä, kuinka se integroituu osaksi ohjelmistokehitystä. Toimeksiantoa ajatellen tarkastelen aihetta erityisesti käyttöliittymätasolta, keskittyen lopulta yhteen keskeisimmistä ohjelmistotestauksen muodoista: yksikkötestaukseen.

Toimeksiantona implementoin Mikkelissä toimivan MHG Systems Oy:n Metsäpaikka -sovellukseen joukon työkaluja, joiden avulla käyttöliittymän yksikkötestausta voidaan automatisoida suurelta osin. Vaikka käyttöliittymätestausta on mahdotonta automatisoida täysin sen inhimillisen luonteen vuoksi, se toimii hyvänä tukena manuaaliselle testaukselle, joka jää usein puutteelliseksi aikarajoitteiden ja lukuisten eri käyttötapausten vuoksi.

Työn tavoitteena on laatia sovelluksen kehittäjille toimiva testausympäristö, ja sitä kautta tarjota heille uusia keinoja virheiden aikaiseen havaitsemiseen ja eliminointiin. Se myös toivon mukaan herättää keskustelua niin asianomaisten kehittäjien, kuin työn

lukijoidenkin keskuudessa siitä, kuinka yksikkötestauksen kautta on mahdollista parantaa - ei pelkästään ohjelman vakautta - vaan myös sitä suorittavan lähdekoodin laatua.

Työssä käytettävät ratkaisut ovat laajalti räätälöity sovelluksen tarpeiden mukaisesti, mutta sen taustalla olevat periaatteet pätevät myös muissa kehitysympäristöissä. Metsäpaikan tapauksessa keskeinen tekniikkavalinta käyttöliittymätasolla on JavaScript-sovelluskehys AngularJS, josta tulen kertomaan tarkemmin luvussa 3.1. Tämän jälkeen luvussa 3.2 esittelen työn toteutuksessa hyödyntämäni testaustyökalut, minkä jälkeen käyn läpi niiden integroimisen varsinaiseen sovellukseen luvussa 4.

2 OHJELMISTOTESTAUS

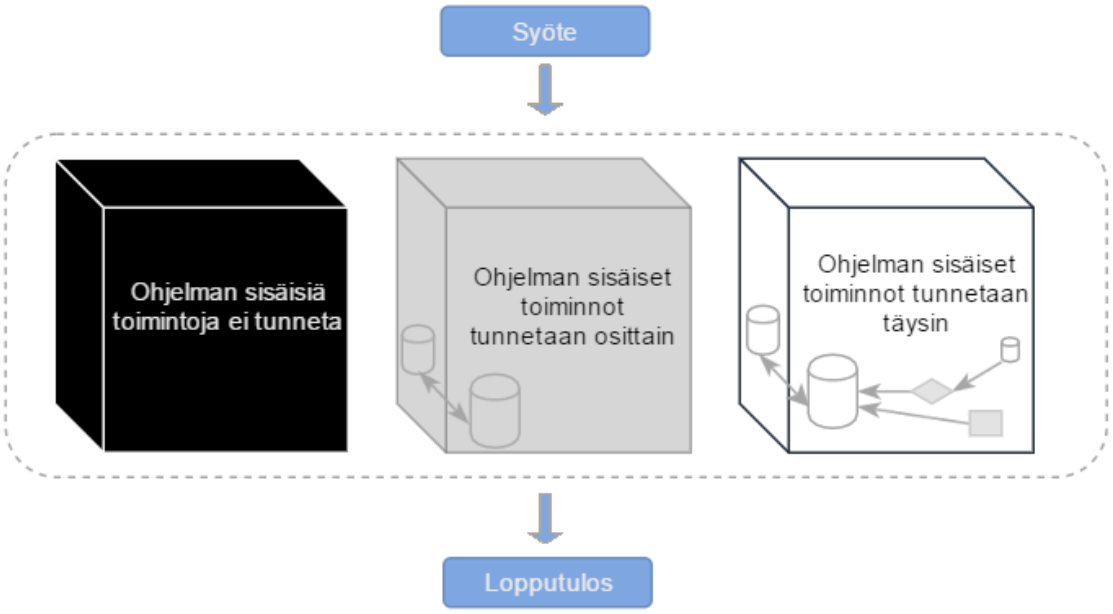
Ohjelmistotestaus - tämä ohjelman laadun varmistamisen keino - on enemmän tai vähemmän järjestelmällinen keino jäljittää ohjelmasta löytyviä vikoja. Kuten Glenford J. Myers (1979, 5) määrittelee: ”Testaus on sellaista ohjelman suorittamista, jossa tarkoituksena löytää siitä virheitä.” Tällä asenteella suoritettu testaaminen on siis parhaimmillaan varsin tuhoavaa toimintaa, sillä sen lopullinen tavoite on rikkoa sovellus tai jokin sen osa. Jos testaajan tavoitteena on puolestaan osoittaa pelkkä ohjelman oikeellisuus, voi syntyä tilanne, jossa virhetilanteita vältetään, jolloin virheiden löytämisestä seuraava lisäarvo jää saamatta. (Korjus 2015, 5.)

2.1 Testaamisen tasot

Jotta testaamisesta saataisiin siis mahdollisimman paljon hyötyä irti, se tulee suorittaa systemaattisesti ja harkiten. Sen sijaan että ohjelmaa testattaisiin satunnaisesti ja vain onnistumisia tavoitellen, testausprosessi jaetaan pienempiin testitapauksiin, joissa myös ohjelman vääränlainen käyttö otetaan huomioon. Testausmetodista riippuen testitapaukset keskittyvät useimmiten ohjelman yksittäisiin komponentteihin tai muihin pienempiin osa-alueisiin. Tällä tavoin testeistä muodostuu kattava kokonaisuus, jonka avulla mahdolliset virhetapaukset voidaan paikantaa vähällä vaivalla. Tarkastelen tässä luvussa eri tasoja, joilla testausta tyypillisesti suoritetaan.

Kun puhutaan jonkin tuotteen testaamisesta, tyypillinen mielikuva testitapahtumasta sisältää usein satunnaisen henkilön kokeilemassa tuotetta käytännössä. Ohjelmistotestauksessa tällaista testausmenetelmää kutsutaan nimellä mustalaatikkotestaus (engl. black-box testing). Kyseisessä testausmuodossa ohjelman sisäisistä toiminnoista ei välitetä, vaan testaaminen keskittyy vain ja ainoastaan siihen, miten ohjelma reagoi suoritettaviin toimenpiteisiin visuaalisesti (Kasurinen 2016, 14).

Koska mustalaatikkotestaus ei vaadi suorittajaltaan ymmärrystä ohjelman teknisistä ratkaisuista, sen voi suorittaa käytännössä kuka tahansa. Testaajan ei tule tietää miten ohjelma *suorittaa* asioita, vaan miten ohjelman tulee *reagoida* tiettyihin tapahtumiin. Tyypillisiä testauksen kohteita ovatkin tästä syystä erilaiset ohjelman elementit, joiden kanssa käyttäjä voi olla vuorovaikutuksessa, kuten napit ja lomakkeet. Koska mustalaatikkotestaus on luonteeltaan varsin mekaaninen, on hyvin tyypillistä, että se automatisoidaan suurelta osin toimenpiteen tehostamiseksi (Kasurinen 2016, 16). Tarkastelen testaamisen automatisointia lisää luvussa 2.3.



KUVA 1. Eri testaustasot havainnollistettuna

Toisin kuin mustalaatikkotestaus, lasilaatikkotestaus (engl. white-box testing) keskittyy ohjelman testaamiseen lähdekooditasolla. Sen lisäksi, että testaajan tulee tietää miten ohjelma toimii, tulee hänen myös ymmärtää miten sitä suorittava koodi toimii. (Kasurinen 2016, 18.) Tyypillisesti lasilaatikkotestausta suorittava henkilö onkin tästä syystä

kehittäjä itse. Lasilaatikkotestauksen suorittaminen toimii myös hyvänä kertauksena kehittäjälle itselleen ja auttaa vahvistamaan hänen ymmärrystä järjestelmän sisäisistä yksityiskohdista.

Lasilaatikkotestaus onkin keskeinen osa päivittäistä kehitysprosessia, sillä ohjelmaan liittyvien ongelmatilanteiden ratkominen alkaa lähes aina sen lähdekoodista. Tästä syystä myös lasilaatikkotestausta on suotavaa automatisoida mahdollisimman paljon, sillä sen avulla pienimmätkin ongelmat on mahdollista havaita heti, kun ongelmakoodi kirjoitetaan. Tyypillisin lasilaatikkotestauksen muoto on yksikkötestaus, josta kerron tarkemmin luvussa 2.3.

Kolmas testaustyyppi - harmaalaatikkotestaus (engl. grey-box testing) - on yhdistelmä musta- ja lasilaatikkotestauksesta. Siinä yhdistyvät mustalaatikkotestauksen pinnallinen testaustyyli, sekä osittainen tuntemus sitä suorittavan koodin toiminnasta. Tyypillinen harmaalaatikkotestauksen kohde on ominaisuus, joka hyödyntää jotain kolmannen osapuolen tarjoamaa rajapintaa tiedon noutamiseksi. Tässä tapauksessa rajapinnan taustalla olevat toiminnot ovat testaajalle saavuttamattomissa, jolloin sovellusta voidaan oman koodin osalta tarkastella lasilaatikkotasolla, mutta hyödynnettävän rajapinnan toimintaa voidaan vain tulkita mustalaatikkotasolta. (Kasurinen 2016, 19–20.)

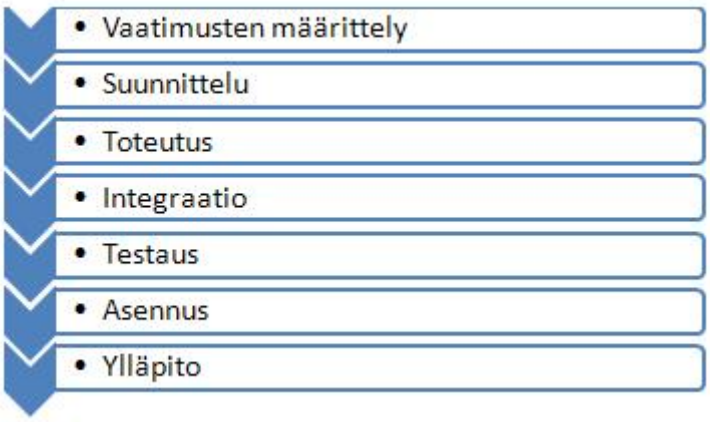
2.2 Testaaminen osana ohjelmistotuotantoa

Ohjelmiston testaaminen on oleellinen osa sen tuotantoprosessia, oli kyseessä mikä tuotantomenetelmä tahansa. Se missä vaiheessa projektia, miten, ja miksi testausta suoritetaan, riippuukin juuri käytetystä menetelmästä. Tarkastelen tässä luvussa yleisiä ohjelmistotuotannon menetelmiä, ja erityisesti sitä, kuinka testaus integroituu niihin.

Vesiputousmalli

Yksi perinteisimmistä ohjelmistotuotannon malleista on nimeltään vesiputousmalli. Se saa nimensä sen kaavamaisesta lähestymistavasta ohjelmistotuotannon prosessiin, jossa sen eri vaiheet etenevät järjestyksessä alaspäin, kuten vesi vesiputouksessa (kuva 2). Tälle lähestymistavalle ominaista on se, että projektin alussa laadittua suunnitelmaa ei

muuteta lähes ollenkaan projektin aikana, vaan se käydään läpi sellaisenaan vaihe vaiheelta.

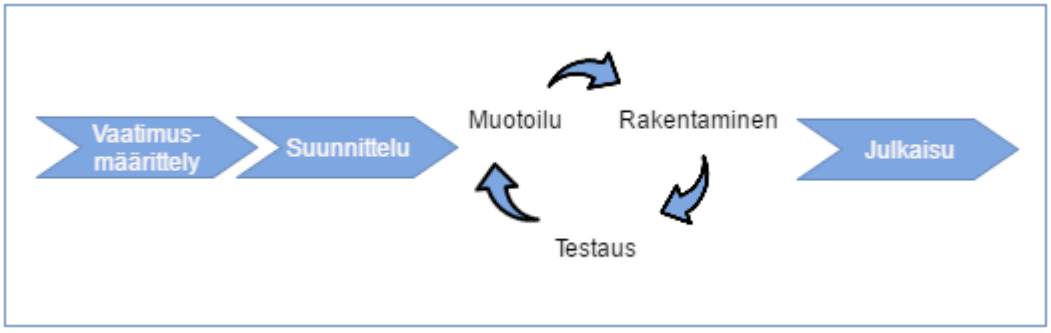


KUVA 2. Vesiputousmallin eri vaiheet

Vesiputousmallissa testauksen rooli on suhteellisen pieni tuotekehityksen näkökulmasta. Testaus sijoittuu elinkaaressa varsinaisen toteutuksen jälkeiseen ajanjaksoon, eli sen tarkoituksena on käytännössä vain varmistaa, että ohjelma toimii suunnitellusti. Toisin kuin ketterämmissä ohjelmistotuotannon menetelmissä, vesiputousmallissa testauksella ei pyritä parantamaan ohjelmaa konseptitasolla, sillä vaatimusmäärittelyjä ei ole tarkoitus muuttaa alussa tehtyjen päätösten jälkeen. (Tietojenkäsittelytieteen laitos 2009, 3–5.) Tämän vuoksi kyseinen malli ei sovellu suureen osaan projekteista, sillä on hyvin yleistä, että projektin alkuvaiheessa ei vielä täysin tiedetä, mitä lopullinen tuote tulee sisältämään.

Ketterä ohjelmistokehitys (engl. agile software development)

Ohjelmistokehityksessä on erityisesti viimeisen vuosikymmenen aikana noussut suosioon kehitysmenetelmiä, jotka poikkeavat perinteisestä vesiputousmallista huomattavasti. Ennalta määrätyn, kaavamaisen prosessin sijaan kehitys tapahtuu iteratiivisesti (”ketterästi”) lyhyissä sykleissä, joiden kautta kehitystä on mahdollista uudelleenohjata tarpeen mukaan (kuva 2). Syklit ovat tyypillisesti 1–4 viikon mittaisia projekteja, joiden aikana ohjelmaa kehitetään yhteisellä päämäärällä. Projektin aikana kehitystiimi käy läpi ohjelmistoprojektille ominaiset vaiheet aina suunnittelusta alkaen, päämäärän ollessa julkaistavissa oleva tuote tai ominaisuus. (Tolvanen 2013.)



KUVA 3. Ketterän kehityssyklin vaiheet

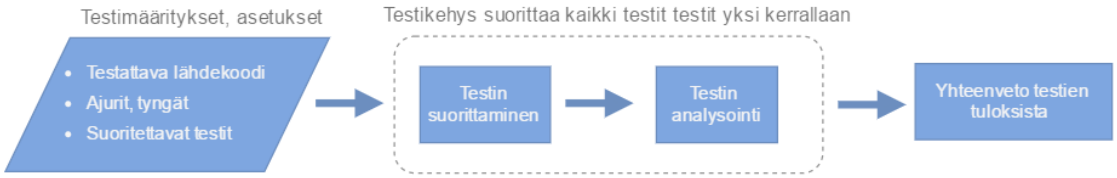
Koska projektin eri vaiheet toistuvat kehityksessä jatkuvasti, testauksen merkitys kasvaa huomattavasti. Sen sijaan, että testausta suoritettaisiin pelkästään ohjelman toimivuuden varmistamiseksi, sen kautta voidaan tehdä konkreettisia havaintoja tuotteen konseptin toimivuuteen liittyen. Tämä mahdollistaa sen, että mahdolliset ongelmat tai puutteet voidaan havaita aikaisin, ja niihin myös kyetään reagoimaan joko silloisessa tai tulevissa kehityssykleissä.

2.3 Yksikkötestaus

Yksikkötestauksella tarkoitetaan kaikessa yksinkertaisuudessaan lähdekoodin yksittäisten osien testausta. Sanalla ”yksikkö” viitataankin juuri tähän; se on jokin koodissa oleva pienempi osa, jonka toiminnallisuutta voidaan testata. (Tietojenkäsittelytieteen laitos 2011, 1.) Käytännössä kyseessä on siis toimenpide, jossa jonkin funktion tai metodin toimivuutta testataan sellaisenaan - eristettynä muusta lähdekoodista. Tämä tapahtuu erillisen ohjelman toimesta siten, että se suorittaa testattavaa lähdekoodia yksikkötestin määrittelemällä tavalla. Kyseistä ohjelmaa kutsutaan nimellä testikehys (test harness). (Korjus 2015, 12.)

Yksittäisen yksikön testaus on kuitenkin usein ongelmallista, sillä lähes aina se on vain pieni osa laajempaa toiminnallisuutta, minkä seurauksesta se saattaa luottaa yhteen tai useampaan yksikköön toimiakseen oikein. Tämän seurauksesta testin kirjoittaja saattaa huomaamattaan päätyä testaamaan useampaa yksikköä samanaikaisesti, jolloin testi lakkaa olemasta yksikkötesti, ja siitä tulee sen sijaan integraatiotesti. Kyseisen ongelman välttämiseksi testattavan yksikön riippuvuuksia joudutaan simuloimaan testiajureiden (test driver) ja tynkien (stub) avulla. (Korjus 2015, 12.)

Se kumpaa milloinkin tarvitsee käyttää, riippuu testattavasta yksiköstä: kutsuuko testattava funktio jotain toista funktiota, vai onko testattavana se osapuoli, jota kutsutaan? Ensimmäisessä tapauksessa yksikön kutsuman funktion tilalle luodaan tynkä, jonka tehtävänä on pohjimmiltaan varmistaa, että se tulee kutsutuksi oikein. Jos kyseinen tynkä palauttaa jotain testattavan yksikön tarvitsemaa dataa, se usein määritellään palautettavaksi etukäteen, jotta voidaan varmistaa, ettei se aiheuta ongelmia yksikön toiminnassa. Vastaavasti jos testattava metodi on se, jota kutsutaan, sitä kutsuvaa metodia voidaan simuloida testiajurin avulla.



KUVA 4. Testitapahtuman vaiheet

Yksikkötestauksesta saatavia hyötyjä on lukuisia, mutta merkittävin niistä on epäilemättä toimenpiteen kertaluontoisuus. Kun yksikkötesti on luotu, siihen ei välttämättä tarvitse kajota lainkaan, ellei sen testaama yksikkö muutu merkittävästi. Fyysisten rajoitteiden poistuessa testi voi kattaa eksponentiaalisen määrän käyttötapauksia samassa ajassa, jonka ihminen tarvitsisi vain yhden tapauksen läpikäyntiin. Tällainen etu tuottaa jo lyhyelläkin aikavälillä huomattavia aikasäästöjä, ja se mahdollistaa yksikön laajan testauksen jokaisella suorituskerralla. Tämä on erityisen tärkeää sellaisen yksikön kohdalla, joka vastaanottaa tietonsa lomakkeen tai muiden epäsäännöllisten käyttäjätoimintojen kautta. (Korjus 2015, 12, MSDN 2016.)

Toinen yksikkötestauksen tarjoama - usein aliarvioitu hyöty on se, että se toimii selkeänä dokumentaationa kehittäjille. Asiallisesti toteutetusta yksikkötestistä on selkeästi nähtävissä kolme yksikön toimintaan liittyvää avainelementtiä: syöte, toimenpide ja odotettu lopputulos. Näiden perustietojen pohjalta tietämätönkin kehittäjä voi pian sisäistää kunkin yksikön käyttötavan ja -tarkoituksen. On myös todettu, että yksikkötestaaminen johtaa paremman koodin kirjoittamiseen, sillä se pakottaa kehittäjän pilkkomaan laajan kokonaisuuden pienempiin, helposti testattaviin osiin. Tämän seurauksesta myös koodin luettavuus paranee ja sitä on entistä helpompi käyttää uudelleen sovelluksen osissa, joissa se muuten jouduttaisiin kirjoittamaan toistamiseen.

Yksikkötestaus sellaisenaan on kuitenkin vain vasta puoli voittoa. Toimiakseen se täytyy suorittaa jonkun toimesta, ja lienee sanomattakin selvää, että sen manuaalinen suorittaminen olisi askel taaksepäin, kun halutaan suoraviivaistaa testausprosessia. Tässä vaiheessa kuvioon astuu aikaisemmin mainitsemani testien automatisointi. Suurin osa saatavilla olevista testikehyksistä tarjoaa kehittäjälle mahdollisuuden määrittää tapahtumia, joiden yhteydessä testit suoritetaan ilman erillistä käskyä. Tämä mahdollistaa muun muassa sen, että testit voidaan asettaa suoritettavaksi aina, kun testattava lähdekoodi, tai suoritettava testi tallennetaan. Tällöin kehittäjä voi havaita rikkovat muutokset lähestulkoon reaaliajassa, oli sitten kyseessä vanhan koodin muokkaus, tai uuden toiminnallisuuden lisäys.

3 KÄYTETTÄVÄT TEKNIIKAT

Esittelen tässä luvussa toimeksiannon kannalta keskeiset sovelluskehykset ja työkalut. Näihin lukeutuvat AngularJS, jolla itse sovellus on kirjoitettu, sekä yksikkötestauksen suorittamisessa käytettävät työkalut Karma ja Jasmine. Havainnollistan kyseisten tekniikoiden toimintaperiaatteita käytännön esimerkkejä hyödyntäen. Esittelyt keskittyvät tekniikoiden perusteisiin, eli lukijalta ei odoteta aikaisempaa kokemusta niiden käytöstä. Huomioitavaa on, että työssä käytetään AngularJS:n versiota 1.5, joka poikkeaa sovelluskehyksen tulevasta 2.0 -versiosta niin paljon, ettei työssä käytetyt esimerkit päde kyseisessä ympäristössä.

3.1 AngularJS

AngularJS on Googlen tukema, avoimeen lähdekoodiin perustuva JavaScript-sovelluskehys. Se sai alkunsa Misko Heveryn ja Adam Abronsin toimesta vuonna 2009 nimellä GetAngular (Austin 2014). GetAngular kehitettiin alun perin sivuprojektina, ja sen oli tarkoitus toimia maksullisena työkaluna web-suunnittelijoille, joilla ei ole kokemusta ohjelmoinnista. Tämä pyrittiin saavuttamaan laajentamalla suunnittelijoillekin tuttua HTML-merkintäkieltä lisäämällä siihen ylimääräisiä käyttöliittymä- ja palvelinpään ominaisuuksia JavaScriptin avulla. (NG-Conf 2014.)

Hylättyään projektinsa liikeideana miesten tiet erkanivat ja Hevery siirtyi Googlen palkkalistoille vuonna 2010 työstämään Google Feedback -sovellusta. Puoli vuotta ja 17000 riviä koodia myöhemmin Hevery kuitenkin turhautui sovelluksen tekniikkavalintoihin ja päätti lyödä vetoa projektin päällikön kanssa, että hän pystyisi uudelleenkirjoittaa sen astisen sovelluksen GetAngularilla huomattavasti yksinkertaisemmin vain kahdessa viikossa. Kolmen viikon panostuksella syntyi samainen sovellus, kirjoitettuna 1500 rivillä koodia. Hevery hävisi siis vedon, mutta vaikuttava lopputulos johti siihen, että GetAngularia alettiin kehittää erillisenä projektina Googlen tukemana, ja näin AngularJS sai alkunsa avoimeen lähdekoodiin perustuvana sovelluskehyksenä. (Austin 2014, NG-Conf 2014.)

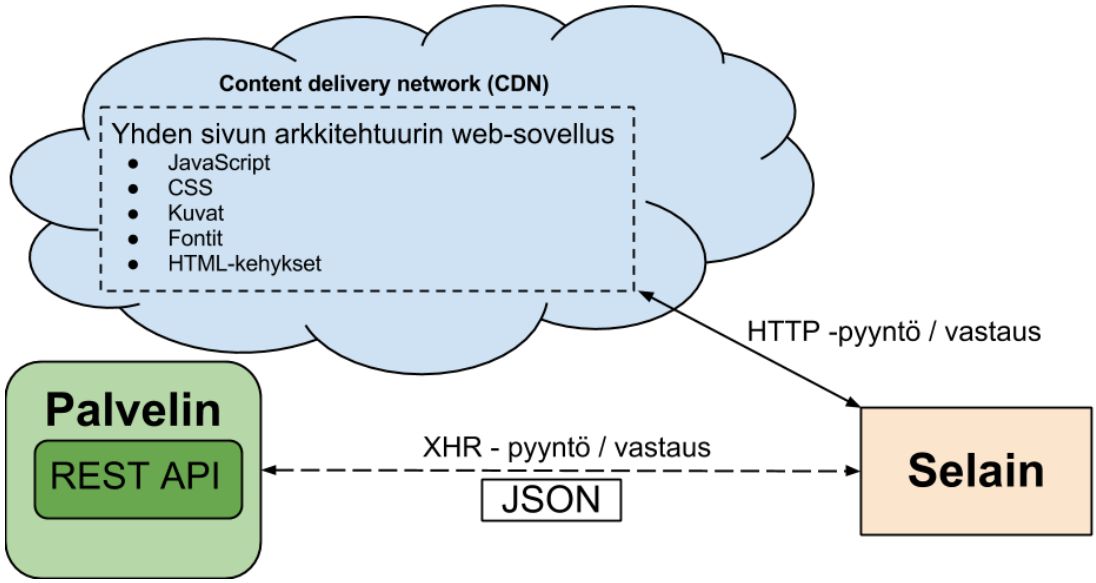
3.1.1 Yhden sivun arkkitehtuuri

AngularJS on siis JavaScript-sovelluskehys, joka mahdollistaa dynaamisten verkkosivujen luomisen yhden sivun arkkitehtuurilla. Tässä luvussa käyn läpi, mitä tämä käytännössä tarkoittaa ja mitä hyötyjä siitä seuraa. Tulen yksinkertaistamisen nimissä puhumaan jatkossa Angularista ilman loppupäätettä ”JS”.

Yhden sivun arkkitehtuurilla tarkoitetaan sitä, että nettisivua ei sen alustavan latauksen jälkeen tarvitse koskaan ladata uudestaan, vaan sisältöä muokataan reaktiivisesti käyttäjän toimintojen perusteella. Tämä saavutetaan siirtämällä kaikki käyttöliittymän suorittamiseen liittyvä toiminnallisuus palvelimelta (server) käyttäjän selaimeen (client). (Viskari 2015.) Käyttäjän saapuessa sivustolle palvelin vastaanottaa ensimmäisen HTTP-pyynnön, jonka vastauksena palautetaan sivuston alustava, usein lähes tyhjä, HTML-pohja ja kaikki sen ulkonäköön (CSS) ja toiminnallisuuteen (JS) liittyvät tiedostot. Tämän jälkeen selain käy läpi vastaanotetut JS-tiedostot, joiden määrittelyiden mukaisesti se sitten täydentää sivuston näkymän.

Huomattavimmat tästä aiheutuvat vaikutukset lienevät havaittavissa käyttöliittymän käytettävyydessä. Kaikki toimenpiteet aina navigoinnista tiedon tallentamiseen tapahtuvat huomattavasti sulavammin kuin perinteisessä useamman sivun arkkitehtuurissa, koska asiakkaan ei tarvitse odottaa, että palvelin palauttaa sille uuden näkymän. (Shimanovsky 2015.) Tästä pääsemme epäsuorasti toiseen merkittävään hyötyyn, eli palvelimen tehokkuuden paranemiseen. Kun palvelimen tehtävistä poistetaan käyttöliittymän

muodostaminen ja lähettäminen, voi se keskittyä hoitamaan vain ja ainoastaan sen merkittävimmän tehtävän, eli tiedonkäsittelyn. Tämä kiihdyttää pyyntöjen läpivientiä, joka puolestaan parantaa käyttäjäkokemusta entisestään.



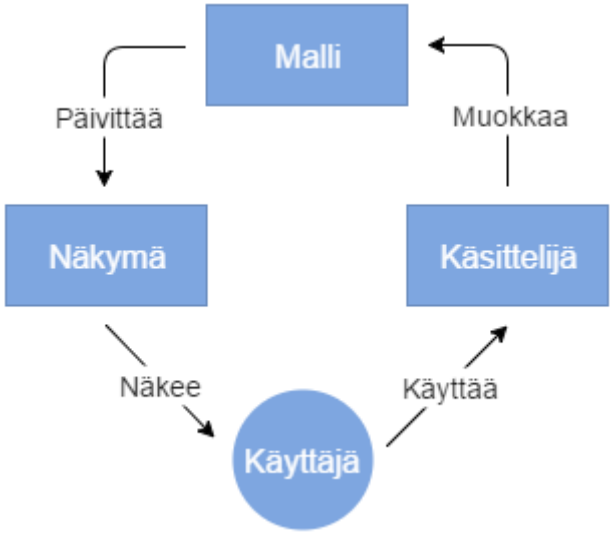
KUVA 5. ”Yksinkertaistettu malli yhden sivun arkkitehtuurilla toteutusta web-sovelluksesta” (Lyytinen 2013)

Kuinka tieto sitten liikkuu yhden sivun arkkitehtuurissa? Tyypillisesti tämä saavutetaan hyödyntämällä kaikkien modernien selainten tukemaa ohjelmointirajapintaa, XMLHttpRequestia (XHR), yhdessä REST-arkkitehtuuria (Representational State Transfer) noudattavan palvelimen kanssa (kuva 5). Yhdessä kyseiset tekniikat mahdollistavat HTTP-pyyntöjen lähettämisen ja vastaanottamisen muulloinkin kuin vain sivunlatauksen yhteydessä (MDN 2016). Tästä seuraa se, että sivustolla tarvittavaa tietoa ei lähes koskaan haluta tai tarvitse hakea yhdellä kertaa, vaan se haetaan pienemmissä palasissa sitä mukaa kuin kyseistä tietoa tarvitaan. Tällöin pyyntöjen käsittelyajat pysyvät hyvin lyhyinä, eikä sivuston käyttö täten esty pitkäksi aikaa kerralla.

3.1.2 MVC

Angularin rakenne perustuu MVC-arkkitehtuuriin. MVC on ohjelmistosuunnittelussa hyödynnetty arkkitehtuurimalli, jota on käytetty erityisesti käyttöliittymäkehityksessä. Sen perusajatuksena on jakaa käyttöliittymän toiminnallisuus kolmeen osaan: malliin (model), näkymään (view) ja käsittelijään (controller). Tällä pyritään saavuttamaan mo-

dulaarinen, pienistä osista koostuva kokonaisuus, jossa kukin osa on helposti muokattavissa tai uudelleenkäytettävissä. (MDN 2015.) Tiedonkulku MVC-arkkitehtuurissa tapahtuu kuvan 6 osoittamalla tavalla. Käyn seuraavaksi läpi, kuinka tämä näkyy käytännössä Angular-sovelluksessa.



KUVA 6. Tiedonkulku MVC-rakenteessa

Angular-sovelluksessa mallia edustaa jokin tieto, joka voidaan sitoa näkymään, ja jonka kanssa käyttäjä voi olla vuorovaikutuksessa. Yksinkertainen malli voisi siis pitää sisällään vain esimerkiksi pienen tekstinpätkän tai numeron. Useimmiten malli kuitenkin on jokin JS-objekti tai lista, joka pitää sisällään useita eri tiedonpalasia.

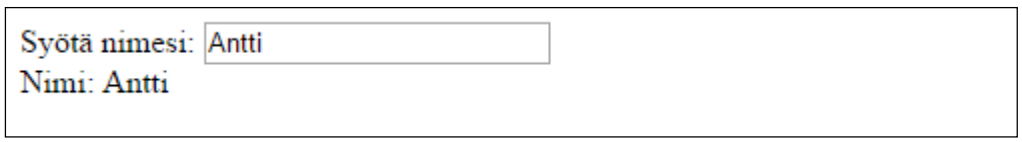
Kun malli on olemassa, se voidaan sitoa MVC:n toiseen osaan, eli näkymään. Näkymä on nimensä mukaisesti se osa sivustoa, jonka käyttäjä näkee. Se koostuu DOM-elementeistä, aivan kuten mikä tahansa muu HTML-pohjainen sivusto, mutta Angularissa tätä vakiotoiminnallisuutta voidaan jatkaa sen omilla ominaisuuksilla, hyödyntämällä HTML:n kustomoitavia attribuutteja. Angularin vakiona tarjoamat attribuutit ovat syntaksiltaan ng-alkuisia, ja ne sidotaan suoraan HTML-elementteihin. Jos tarkoituksena on yksinkertaisesti sitoa tietoa mallista näkymään, se saavutetaan käärimällä kyseisen tiedon muuttuja aaltosulkeisiin. Kuva 7 havainnollistaa edellä mainitut käyttötapaukset.

```
<div ng-app ng-init="nimi = Antti">
    <div>
        Syötä nimesi: <input type="text" ng-model="nimi">
    </div>
    <div>
        Nimi: {{ nimi }}
    </div>
</div>
```

KUVA 7. Mallin sitominen näkymään, havainnollistettuna yksinkertaisella esimerkillä.

Edellä olevassa esimerkissä Angular-toiminnallisuus alustetaan määrittämällä ulommaisimpaan HTML-elementtiin attribuutti ”ng-app”. Tämä kertoo Angularille, että kyseessä on sille kuuluva HTML-malli, jolloin se osaa yhdistää loputkin Angular-määrittelyt samaan ympäristöön. Alustuksen jälkeen ympäristöön luodaan sen ensimmäinen malli komennolla ”ng-init”, joka on käytännössä funktio, jonka sisältö suoritetaan sivun alustusvaiheessa. Funktioissa määritelty malli, ”nimi”, sidotaan myöhemmin kahteen paikkaan alustavalla arvolla ”Antti”.

Ensimmäisen näistä on tavallinen input-elementti, johon malli sidotaan attribuutilla ”ng-model”. Tämä luo DOM-elementin ja mallin välille kaksisuuntaisen yhteyden, mikä tarkoittaa, että yhden arvon muuttuessa muutos näkyy myös toisessa päässä. Seuraavassa div-elementissä samaisen mallin arvo sidotaan näkymään tavallisena tekstinä. Tässä tapauksessa yhteys kulkee vain yhteen suuntaan - mallista näkymään - mutta mallissa tapahtuvat muutokset näkyvät myös kyseisessä elementissä. Jos nimeä siis muutettaisiin input-kentässä, muutokset olisivat heti nähtävissä myös alla olevassa tekstielementissä.



KUVA 8. Koodin lopputulos selaimessa.

Tähänastinen esimerkkini on varsin yksinkertainen, eikä siitä olisi paljoa iloa kenellekään sellaisenaan. Laajentaaksemme sovelluksen toiminnallisuutta, tarvitsemme MVC:n kolmannen osan, eli käsittelijän. Angularissa tämä kulkee samaisella englannin

kielen nimellä controller, ja se luodaan JS-koodin puolella sille tarkoitetulla rakentaja-metodilla. Ennen käsittelijän lisäämistä meidän tulee kuitenkin alustaa Angular-koodiamme hieman tarkemmin.

```
1  var sovellus = angular.module('sovellus', []);
2
3  sovellus.controller('kasittelija', kasittelijaFunktio);
4
```

KUVA 9. Angular-moduulin alustus ja käsittelijän sitominen siihen.

Kuvassa 9. luomme ensimmäisellä rivillä pohjan Angular-ympäristöllemme, johon voimme myöhemmin lisätä ominaisuuksia. Tarkalleen ottaen luomme Angular-moduulin nimeltä ”sovellus”, ja lisäämme sen samalla myös samannimiseen muuttujaan, jotta säästymme pieneltä vaivalta tulevilla riveillä. Moduulin rakentajametodi (angular.module) vastaanottaa nimen lisäksi yhden parametrin, joka on lista moduulin riippuvuuksista.

Riippuvuuksiksi katsotaan sellaiset moduulit tai tehdasfunktiot, jotka sisältävät lähdekoodin tarvitsemaa toiminnallisuutta. Ne ovat myös erinomainen keino säilöä logiikkaa ja metodeja, joita halutaan käyttää useassa eri oliossa. Useimmiten nämä ovat siis jonkin kolmannen osapuolen laatimia koodikirjastoja tai komponentteja, joiden tarkoituksena on lisätä projektiin toiminnallisuutta jota Angular ei tarjoa vakiona, tai jonka toteuttaminen itse olisi tarpeettoman työlästä. Tässä tapauksessa emme kuitenkaan tarvitse ylimääräisiä ominaisuuksia, joten jätämme listan tyhjäksi.

Moduulin alustettuamme voimme lisätä siihen käsittelijän aikaisemmin mainitsemallani rakentajametodilla angular.controller. Rivillä 3 syötämme controller -metodille parametrinä sen nimen ”kasittelija”, sekä funktion ”kasittelijaFunktio”, jonka avulla voimme lisätä tarvitsemamme mallit ja metodit käsittelijään. Jotta kyseiset lisäykset olisivat sidottavissa myös näkymään, ne täytyy ensiksi lisätä Angularin skooppi -olioon ($scope) kuvan 10 osoittamalla tavalla. Olion nimen alussa oleva dollari-merkki viittaa siihen, että se kuuluu Angularin vakio-ominaisuuksiin. Jos haluaisimme luoda oman tehdasfunktion (angular.factory), sen ominaisuudet olisi mahdollista tuoda käsittelijään kuvassa käytetyn $inject -metodin avulla.

```
5  kasittelijaFunktio.$inject = ['$scope'];
6
7  function kasittelijaFunktio($scope) {
8      $scope.nimi = 'Antti';
9  }
```

KUVA 10. Käsittelijäfunktion määritys, ja ominaisuuksien lisäys Angular skooppiin.

Se miksi joudumme käyttämään $scope -oliota, johtuu Angularin tavasta sitoa sovelluksen näkymä sen muihin osiin. Kun sidomme Angular-toiminnallisuutta näkymäämme (esimerkiksi kuvassa 7. käytetty ng-model -attribuutti), selain ei osaa tulkita sitä sellaisenaan, koska se ei sisällä mitään logiikkaa HTML:n näkökulmasta. Sen sijaan sivun alustusvaiheessa Angularin vastuulle jää näiden ominaisuuksien tulkinta. Tämän se saavuttaa etsimällä HTML-mallista Angular-toiminnallisuutta sisältävät elementit ja käärimällä ne jQuery-objekteiksi. Kyseiset objektit sisältävät vakiona lukuisia DOM-manipulointiin liittyviä metodeja, mutta sen tärkein ominaisuus on yhteys sitä ympäröivään skooppiin. Jotta näillä objekteilla olisi käytössään juuri haluamamme skooppi, se tulee määritellä erikseen HTML-attribuutilla.

```
9  <div ng-app="sovellus" ng-controller="kasittelija">
10     <div>
11         Syötä nimesi: <input type="text" ng-model="nimi">
12     </div>
13     <div>
14         <span id="teksti">Nimi: {{ nimi }}</span>
15     </div>
16 </div>
```

KUVA 11. Sovelluksen näkymä, joka on sidottu tiettyyn moduuliin ja käsittelijään.

Kuvassa 11. teemme tarvittavat lisäykset, jotta sovellus tulee kytketyksi oikeaan skooppiin. Erot aikaisempaan esimerkkiin (kuva 7) ovat käytännössä rivillä yhdeksän, jossa määrittelemme nimettömän sovelluksen (ng-app) sijaan käyttämämme ”sovellus”-moduulin. Tämän seurauksesta sillä on myös käytettävissä äsken luomamme käsittelijä, jonka syötämme elementille ng-controller-attribuutilla. Samalla poistimme aiemmin käyttämämme ng-init-attribuutin, jota emme tarvitse enää, koska siirsimme sen sisällön käsittelijään. Varsinainen lopputulos ei siis poikkea aikaisemmasta mitenkään, mutta nykyinen järjestely mahdollistaa monimutkaisemman logiikan lisäämisen sivulle. Tämä

saavutettaisiin käytännössä lisäämällä metodeja suoraan käyttämämme käsittelijän skooppiin, samalla tyylillä kuin määrittelimme ”nimi”-mallinkin.

3.2 Testaustyökalut ja -sovelluskehykset

Samalla tavalla kuin Angular ja muut vastaavat sovelluskehykset on luotu helpottamaan käyttöliittymän luomista, myös testauspuolelta löytyy omat vastineensa. Vaikka koodin testaaminen ilman erillisiä työkaluja on toki mahdollista, se on loppujen lopuksi varsin tehotonta ja aikaa vievää. Tämän vuoksi testaamisen tueksi on jo vuosikymmenien ajan ohjelmoitu sitä helpottavia työkaluja. Esittelen tässä luvussa varsinaisessa toteutuksessa käyttämäni testaustyökalut - Jasminen ja Karman - jotka yhdessä muodostavat puitteet testien suorittamiselle.

3.2.1 Jasmine

Jasmine on toinen työssä käytettävistä testaussovelluskehyksistä, joka - kuten Karmakin - perustuu avoimeen lähdekoodiin. Sen rooli testausarkkitehtuurissa on keskeinen, sillä se tarjoaa testikoodillemme helposti ymmärrettävän rakenteen ja syntaksin. Tämän se saavuttaa implementoimalla kattavan joukon metodeja, joiden toiminnallisuus ja nimet on suunniteltu siten, että oikein käytettynä ne muodostavat eräänlaisen tarinan ohjelmoijan testikoodiin. Jasminen virallisen dokumentaation mukaan (2015) sen muita vahvuuksia ovat riippumattomuus muista sovelluskehyksistä, sekä se, ettei se vaadi DOM:ia toimiakseen.

Havainnollistaakseni Jasminen syntaksin intuitiivisuutta, käyn läpi sen keskeisimmät osat pala palalta. Koska Jasminen vahvuus on sen kuvaavassa syntaksissa (joka luonnollisesti koostuu englannin kielen sanoista), käytän yhtenäisyyden nimissä Englantia myös niissä osissa, missä se ei välttämättä ole vaadittua. Syy tälle lienee havaittavissa jo seuraavassa koodinpätkässä.

```
1  describe('Demonstrating Jasmine syntax', function() {
2      it('should show how to run a simple test', function() {
3          var number = 2;
4          expect(number).toBe(2);
5      });
6  });
```

KUVA 12. Yksinkertainen esimerkki Jasmine-testin rakenteesta

Kuvassa 12. demonstroin Jasminella tehdyn testin rakennetta. Se alkaa riviltä 1 kutsumalla describe-metodia, jonka tarkoitus on nimensä mukaisesti kuvailla pian suoritettavia testejä yleisesti. Se vastaanottaa kaksi parametriä, joista ensimmäinen on string-muotoinen kuvaus, ja toinen on funktio, joka sisältää varsinaiset testit. Sen lisäksi että tämä rakenne helpottaa testin luettavuutta, se mahdollistaa myös sen, että testin epäonnistuessa ohjelmoijalle voidaan selkeästi ilmoittaa, missä vaiheessa testi meni pieleen. Tämä on erityisen tärkeää laajoissa sovelluksissa, joissa testien määrä saattaa kavuta satoihin tai tuhansiin.

Testin toinen vaihe, eli esimerkin rivillä 2. suoritettava metodi ”it” on näennäisesti hyvin samankaltainen kuin edellinenkin vaihe. Se vastaanottaa kuvauksen sekä suoritettavan funktion, mutta tässä tapauksessa funktio onkin varsinainen suoritettava testi. Testin sisällä voi suorittaa lukuisia asioita, kuten etsiä DOM:ista elementtejä ja arvoja, tutkia mallien sisältöjä tai suorittaa funktioita ja tarkastella niiden vaikutuksia edellä mainittuihin asioihin. Tulemme törmäämään näihin eri tapauksiin myöhemmin toteutusosassa, mutta kuvan 12 esimerkissä suoritettava testi keskittyy vain yksinkertaisen mallin tutkimiseen.

Rivillä 3 määrittelemme muuttujan ”number”, joka saa arvokseen luvun kaksi. Seuraavalla rivillä aloitamme testin suorittamisen kutsumalla metodia ”expect”, joka vastaanottaa tutkittavan arvon, ja vertaa sitä toiseen arvoon, jonka syötämme ketjutettuun metodiin ”toBe”. Testi luokitellaan onnistuneeksi, jos kyseiset arvot vastaavat toisiaan. Vastaavasti jos arvot poikkeavat toisistaan, testinsuorittaja ilmoittaa virheestä hyödyntämällä kaikkia testiin liittyviä kuvauksia (describe, it) sekä varsinaista suoritettua testilauseketta (expect, toBe). Tällöin käy ilmi mikä testi on kyseessä, ja miksi se epäonnistui.

Kun tarkastelemme testiä kokonaisuutena, siitä on selkeästi havaittavissa tietynlainen selkeys ja luonnollisuus, joka ei ole kovin tyypillistä ohjelmoinnissa. Testin eri vaiheet on ilmaistu niin luonnollisella kielellä, että käytännössä kuka tahansa voi ymmärtää mitä testillä tavoitellaan. Tämä on myös loistava havainnollistus siitä, mitä mainitsin dokumentointiin liittyvistä hyödyistä luvussa 2.3. Kun kehittäjä lukee toisen tekemää

koodia, on sen sisäistäminen huomattavasti nopeampaa ja helpompaa, jos sille on kirjoitettu testit Jasminen kaltaisella sovelluskehyksellä.

3.2.2 Karma

Karma on Angular-kehitystiimin luoma työkalu testien suorittamiseen. Sen tavoitteena on tarjota kehittäjälle mahdollisimman helppo tapa suorittaa kirjoitetut yksikkötestit, kuitenkaan rajoittamatta häntä tiettyjen työkalujen käyttöön. Tästä syystä Karmalla voi suorittaa käytännössä millä tahansa testaussovelluskehyksellä kirjoitettuja testejä, ja Jasmine on yksi näistä. (Karma 2016.)

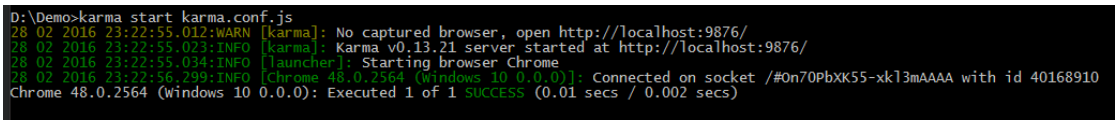
Karma toimii käytännössä vertailemalla testattavan sovelluksen lähdekoodia kirjotettuun testikoodiin. Se tekee tämän käynnistämällä paikallisen web-palvelimen, jonka avulla se suorittaa koodin määritellyissä selaimissa. Toimiakseen Karma vaatii kuitenkin tiedon edellä mainituista asioista, ja nämä tiedot saadaan parhaiten välitettyä erillisen asetustiedoston avulla. (Karma 2014.)

```
1  module.exports = function(config) {
2    config.set({
3      frameworks: ['jasmine'],
4
5      files: ['**/*.js'],
6
7      autoWatch: true,
8
9      browsers: ['Chrome']
10   })
11 }
12
```

KUVA 13. Esimerkki Karman asetustiedostosta.

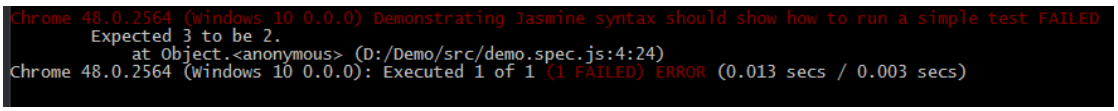
Kuvassa 13. määrittelemme minimaalisen määrän asetuksia, joiden avulla saamme Karman toimimaan haluamallamme tavalla. Aloitamme riviltä kolme määrittelemällä käyttämämme testisovelluskehyksen, jolla testit on kirjoitettu. Tämän jälkeen rivillä 5 luomme listan, joka sisältää polut suoritettaviin tiedostoihin. Tässä tapauksessa käyttämämme polkumäärittely etsii kaikki JavaScript-tiedostot siitä kansiosta, jossa asetustiedosto sijaitsee, sekä myös kaikista sen alakansioista. Rivillä 7 määrittelemämme ”autoWatch” -asetus kertoo Karmalle, että testit suoritettuaan se jää kuuntelemaan edellä

mainittuja tiedostoja muutosten varalta. Jos jotain määritellyistä tiedoista muokataan, Karma suorittaa testit automaattisesti uudelleen. Lopuksi rivillä 9 määrittelemme listan selaimia, joissa testit tulee suorittaa. Tässä tapauksessa käytämme vain Google Chromea.



KUVA 14. Testien suorittaminen Karmalla

Testien suorittaminen Karmalla tapahtuu komentorivillä komennolla ”karma start”, ja sille annetaan lisäksi polku äsken määriteltyyn asetustiedostoon (kuva 14). Koska kyseinen tiedosto on tässä tapauksessa samassa kansiossa, kuin mistä komento suoritetaan, riittää poluksi pelkkä tiedoston nimi. Käynnistyksen jälkeen Karma ilmoittaa käynnistävänsä haluamamme selaimen testien suoritusta varten, minkä jälkeen se suorittaa varsinaiset testit. Käytin tässä esimerkissä edellisessä luvussa luomaani testiä, joka tutkii onko määrittelemämme muuttujan arvo luku kaksi (kuva 8). Testit suoritettuaan Karma ilmoittaa viimeisellä rivillä testien läpäisystä.



KUVA 15. Ilmoitus epäonnistuneesta testistä

Vaihtamalla testissä olevan muuttujan arvoa (tässä tapauksessa luvuksi kolme) näemme, kuinka Karma ilmoittaa epäonnistuvasta testistä (kuva 15). Se aloittaa ilmoittamalla epäonnistuvan testin nimen (description), jonka jälkeen se kertoo, mikä tarkasteltavan muuttujan arvo oli, sekä mikä sen odotettiin olevan. Tämän jälkeen se ilmaisee testin tarkan sijainnin kertomalla polun kyseiseen testitiedostoon, sekä miltä riviltä epäonnistuva testi löytyy.

4 CASE METSÄPAIKKA

Metsäpaikka on MHG Systems Oy:n kehittämä sovellus metsänomistajille, ja sen tarkoitus on yksinkertaistaa metsänhoitoon liittyviä toimenpiteitä. Näihin lukeutuvat muun

muassa metsätilan tietojen luominen, päivittäminen ja jakaminen. Sovelluksesta on olemassa sekä mobiili- että selainversio, joista jälkimmäistä olen itse ollut mukana kehittämässä keväästä 2015 alkaen. Sovelluksen aktiivinen kehitys alkoi muutama viikko ennen kuin liityin kehitystiimiin.

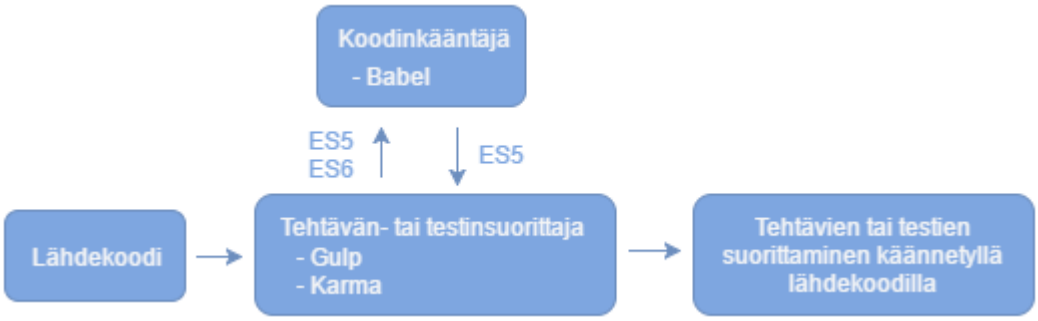
Vaikka sovellusta oli tämän projektin alkaessa kehitetty jo useita kuukausia, käyttöliittymän toimintaan liittyvää testiautomaatiota ei ollut vielä otettu huomioon. Sen sijaan tarvittava testaus suoritettiin aina manuaalisesti, mikä osoittautui lyhyelläkin aikavälillä varsin työlääksi ja itseään toistavaksi toimenpiteeksi. Opinnäytetyöni toimeksiantona tehtäväni oli minimoida tällaisen testauksen tarvetta testiautomaation kautta. Käyn tulevissa luvuissa läpi, kuinka integroin testaustyökalut sovellukseen, sekä demonstroin niiden käyttöä käytännön esimerkkien kautta.

4.1 Testiympäristön luominen

Käytettävien testaustyökalujen valitseminen oli suhteellisen suoraviivainen toimenpide. Jo lyhyen tutkinnan jälkeen oli ilmeistä, mitä työkaluja Angular-yhteisö suosii yksikkötestauksen toteuttamiseksi. Näihin lukeutuvat luvussa 3.2 esittelemäni Jasmine ja Karma. Testiympäristön luominen tarjosi kuitenkin omat haasteensa, sillä työkalut lisättiin olemassa olevaan projektiin, minkä vuoksi niiden tuli myös myötäillä muita projektissa käytettyjä työkaluja. Työssä käytettyjen työkalujen käyttö ja niiden asennus olettaa, että koneelta löytyy asennettuna JavaScriptiä palvelimella suorittava NodeJS, sekä sen mukana tuleva paketinhallintajärjestelmä npm (node package manager).

Metsäpaikassa keskeisin käyttöliittymäkehityksessä hyödynnetty työkalu on nimeltään Gulp. Se on NodeJS:n avulla toimiva tehtävänsuorittaja, jonka avulla on mahdollista automatisoida usein toistettavia tehtäviä, jotka olisi varsin työlästä toteuttaa manuaalisesti. Työn kannalta oleellinen Gulp-toimenpide sisältää lähdekoodin kääntämistä yhdeltä kieleltä toiselle. Tarkalleen ottaen kyseessä on tulevan JavaScript-version - ECMAScript 6:n (ES6) - käyttäminen nykyisen ECMAScript 5:n (ES5) lisäksi. Koska nettiselaimet eivät vielä tue uutta versiota täysin, alkuperäinen koodi tulee kääntää muotoon, jota kaikki selaimet ymmärtävät. Tämä saavutetaan kuvan 16. osoittamalla tavalla, hyödyntäen toista kolmannen osapuolen kirjastoa nimeltä Babel, jonka tehtävä on havaita lähdekoodista uutta syntaksia hyödyntävät koodinpätkät, ja kääntää ne nykyisen

JavaScript-version mukaiseksi. Koska Karma toimii irrallaan Gulpista, kyseinen käännösprosessi tulee integroida siihen erikseen.



KUVA 16. Lähdekoodin kääntäminen suoritettavaan muotoon

Nämä asiat huomioon ottaen voimme siirtyä asentamaan tarvittavia testaustyökaluja. Asennus tapahtuu kätevästi suoraan komentorivin kautta, hyödyntäen äsken mainitsemaani paketinhallintajärjestelmää (npm). Koska yksikkötestaaminen on päällepäin melkoista välineurheilua, asennettavien pakettien lista on tässäkin tapauksessa suhteellisen pitkä. Asennus onnistuu kuitenkin helposti kuvan 17. mukaisesti avaamalla komentorivi projektin kansioon, ja luettelemalla paketit yhdessä letkassa komennon ”npm install” jälkeen.

```
D:\Coding\MHG\trading-portal\src\main\webapp>npm install jasmine jasmine-core
karma karma-babel-preprocessor karma-chrome-launcher karma-jasmine karma-phant
omjs-launcher phantomjs-prebuilt --save-dev
```

KUVA 17. Testaustyökalujen lataus ja asennus npm:n kautta

Ladattuihin paketteihin lukeutuvat Karman ja Jasminen lisäksi PhantomJS (phantomjs-prebuilt), joka on käytännössä ”päätön” selainmoottori, eli se suoritetaan täysin komentorivin kautta ilman graafista käyttöliittymää. Tämä mahdollistaa testien nopean suorittamisen ilman tarvetta erillisen selainohjelman käynnistämiselle. Testit suorittava Karma vaatii kuitenkin erillisen paketin (karma-*selain*-launcher) jokaiselle selaimelle, jossa testit halutaan suorittaa, joten latasin vastaavat paketit PhantomJS:lle, ja varmuuden vuoksi myös Google Chromelle, jos testit halutaan suorittaa useammassa selaimessa. Näiden lisäksi tarvitsin vielä paketin, joka mahdollistaa lähdekoodin kääntämisen Babelin avulla testien suorittamisen yhteydessä (karma-babel-preprocessor).

Pakettien asennuksen jälkeen jäljelle jää Karman asetustiedoston määrittäminen samalla tyylillä, kuin kävin läpi luvun 3.2 esimerkissä. Projektin ollessa huomattavasti esimerkkitapausta monimutkaisempi, myös asetustiedosto tulee sisältämään joitakin uusia määrityksiä aikaisempaan verrattuna. Luotu asetustiedosto, testien käyttämät esimerkkidatat ja muut aputiedostot löytyvät omasta ”test”-kansiosta projektin sisältä.

```
1  module.exports = function(config) {
2    config.set({
3      // Polku (suhteessa conf-tiedostoon), jonka mukaan muut tiedostot haetaan projektista
4      basePath: '../',
5
6      // Muut käytetyt testisovelluskehykset
7      frameworks: ['jasmine'],
8
9      // Lista testeihin ladattavista tiedostoista
10     files: [
11         // Sovelluksessa käytettävät kolmannen osapuolen kirjastot ja sovelluskehykset
12         // Testeissä käytettävät kolmannen osapuolen kirjastot
13         // Lähdekoodit
14         'app/**/*.js',
15         // Suoritettavat testit
16         // Tyngät yms, aputiedostot
17         'test/config/translation.service.mock.js',
18         // Testeissä käytettävät esimerkkidatat (oikeanmallinen JSON)
19     ],
20
21     // Tiedostot, joita ei haluta ladata sivulle vaikka ne täyttäisivät edellä mainitut ehdot
22     exclude: [
23         'app/services/translation.service.js'
24     ],
25
```

KUVA 18. Karman asetustiedosto: käytetyt sovelluskehykset, tiedostot

Kuvassa 18 on asetustiedoston alkuosa, joka sisältää pitkälti samat määritykset kuin aikaisempikin esimerkki. Kuvan lyhentämiseksi jätin varsinaiset tiedostot listaamatta, mutta kommentit sisältävät kuvaukset niistä tiedostotyypeistä, jotka kyseiseen ”files”-listaan tulee määritellä. Uutena asiana tässä on kuitenkin viimeinen ”exclude”-määritys, jonka avulla on mahdollista jättää pois joitakin tiedostoja, vaikka ne täyttäisivät edellisessä vaiheessa määritellyt ehdot. Tässä tapauksessa olen ladannut lähdekoodit poimimalla kaikki ”.js”-päätteiset tiedostot ”app”-kansiosta (rivi 14), mutta haluan testejä suorittaessa korvata yhden tiedoston (rivi 23) erillisellä tynkäversiolla (rivi 17), koska alkuperäisen tiedoston suorittaminen testeissä aiheuttaisi virheitä. Vastaavasti sen täysi puuttuminen aiheuttaisi myös ongelmia, minkä vuoksi sen tilalle ladataan samaa nimeä kantava yksikkö, joka yksinkertaisesti ei tee mitään.

```
24      // Lisäosat, jotka käsittelevät lähdekoodia testien suorittamisen yhteydessä
25      preprocessors: {
26          'app/**/*.js': ['babel']
27      },
28
29      // Babel-lisäosan omat asetukset
30      babelPreprocessor: {
31          options: {
32              presets: ['es2015']
33          },
34          filename: function (file) {
35              return file.originalPath.replace(/\.js$/, '.es5.js');
36          },
37          sourceFileName: function (file) {
38              return file.originalPath;
39          }
40      },
```

KUVA 19. Karman asetustiedosto: käytetyt lisäosat

Seuraavaksi Karma asetetaan hyödyntämään Babelin koodinkääntöominaisuutta määrittelemällä tiedostot, jotka halutaan kääntää testiensuorittamisen yhteydessä (kuva 19). Tämä onnistuu rivin 26 mukaisella määrityksellä, joka käytännössä kertoo Karmalle, että js -päätteiset tiedostot halutaan käsitellä Babelin avulla. Riviltä 30 alkaen määritellään joukko asetuksia kääntäjälle, joissa muun muassa määritellään lähdekoodissa käytetty JavaScript-versio (tässä tapauksessa ES6, toiselta nimeltään ES2015), sekä ohjeet käännettyjen tiedostojen uudelleennimeämiselle testisuorituksen ajaksi.

```
42      // Määritykset sille, kuinka testit raportoidaan
43      reporters: ['progress'],
44      // Karman webbipalvelimen käyttämä porttinumero
45      port: 9876,
46      // Määrittelee värien käytön testiraporteissa
47      colors: true,
48      // Määrittelee minkä tason "loggaukset" näytetään konsolissa
49      logLevel: config.LOG_INFO,
50      // Määrittelee, jääkö Karma kuuntelemaan suoritettavia tiedostoja muutosten varalta
51      // Jos päällä, testit suoritetaan uudelleen muutoksen jälkeen
52      autoWatch: true,
53      // Selaimet, jossa testit suoritetaan (vaatii erillisen launcherin)
54      browsers: ['PhantomJS'],
55      // Suoritetaanko vain kertaalleen
56      singleRun: false,
57      // Määrittää, kuinka monta selainta voidaan käynnistää yhtäaikaisesti
58      concurrency: Infinity
59    })
60  }
61
```

KUVA 20. Karman asetustiedosto: yleiset suoritukseen liittyvät asetukset

Lopuksi asetustiedostossa määritellään yleisiä suoritukseen liittyviä asetuksia kuvan 20 osoittamalla tavalla. Kuvassa olevat kommentit selittävät kunkin määrityksen tarkoitukset, joten en käy niitä erikseen läpi. Huomautettavaa on kuitenkin käytetyn selaimen, PhantomJS:n, määrittely aikaisemmin käytetyn Google Chromen sijaan (rivi 54).

4.2 Testien kirjoittaminen

Testiympäristön ollessa paikallaan, jäljelle jää enää itse testien kirjoittaminen - tai ainakin niin kuvittelin. Testien kirjoittamista aloittaessa kävi pian selväksi, että suurta osaa olemassa olevasta koodista piti muokata, jotta sitä olisi mahdollista testata tehokkaasti yksikkötestin määritelmän mukaisesti. Käytännössä ongelmat aiheutuivat siitä, että yksittäisen toimenpiteen eri vaiheet oli koottu kaikki saman funktion sisälle, jolloin niitä oli mahdotonta testata ilman, että funktion muut vaiheet tulisi suoritetuksi samalla. Tämä on tyypillinen ilmiö sovelluksessa, jota ei ole ohjelmoitu testaus mielessä.

Ongelman ratkaistakseni jouduin siis jakamaan monia olemassa olevia funktioita pienempiin osiin, kuitenkaan muuttamatta niiden perimmäistä toiminnallisuutta. Tällaista koodin muokkaamista kutsutaan refaktoroinniksi, ja sen pääasiallinen tarkoitus on parantaa koodin sisäistä rakennetta. Testattavuuden helpottamisen lisäksi refaktoroinnin kautta saatavat hyödyt ovat havaittavissa muun muassa koodin luettavuuden, ylläpidettävyyden ja uudelleenkäytettävyyden paranemisena. (Ruotsalainen 2014, 29.) Esittelen yhden esimerkin refaktoroinnista luvussa 4.2.1.

Metsäpaikan - kuten käytännössä kaikkien muidenkin Angular-sovellusten - kolme käytetyintä Angular-ominaisuutta ovat tehdasfunktiot, käsittelijät sekä komponentit. Kyseiset ominaisuudet poikkeavat toisistaan käyttötarkoituksen lisäksi siinä, kuinka ne luodaan sivunlatauksen yhteydessä. Tämä tulee ottaa huomioon myös testejä kirjoittaessa, sillä kyseiset yksiköt tulee alustaa erikseen jokaisen testin yhteydessä. Käyn tulevissa luvuissa läpi, kuinka näitä eri ominaisuuksia on mahdollista testata.

4.2.1 Tehdasfunktiot (factory)

Angularin tehdasfunktiot ovat keskeinen tapa jakaa logiikkaa tai dataa sovelluksen sisällä. Toisin kuin käsittelijät, tehdasfunktioita ei ole sidottu mihinkään HTML-

elementtiin, minkä ansiosta ne säilyttävät sisältönsä niin pitkään, kun sivu on ladattuna. Tämän vuoksi ne ovat otollinen paikka säilöä sivuston tarvitsemia malleja ja niihin liittyvää logiikkaa. Kuten mainitsin lyhyesti luvussa 3.1.2, tehdasfunktioiden ominaisuuksien lisääminen käsittelijään tai muihin tehdasfunktioihin onnistuu Angularin $inject-metodilla, ja tulemme käyttämään kyseistä metodia runsaasti myös yksikkötesteissä.

```
function calculateSingle(operation) {
    return $http.get(domain + 'api/ws/rest/operations/' + operation.id + '/value')
        .then(result => {
            operation.price = result.data.value;
            if (operation.type.mainType === 2 && operation.price > 0) {
                operation.price = operation.price * -1;
            }

            var i = _.findIndex(service.operationList, op => {
                return op.id === operation.id
            });

            service.operationList[i] = operation;
            return operation;
        })
        .catch(err => {
            return err;
        });
}
```

KUVA 21. Testattavan metodin calculateSingle alkuperäinen versio

Ennen testin laatimista, esittelen testattavan yksikön (kuva 21). Kyseessä on OperationService -nimisen tehdasfunktion metodi calculateSingle, jonka tehtävä on hakea HTTP-pyynnöllä rahallinen arvo sen vastaanottamalle operaatio-objektille. Haun jälkeen arvo lisätään objektiin joko positiivisena tai negatiivisena sen perusteella, onko kyseinen operaatio tyypiltään (type.mainType) 1 vai 2. Lopuksi operaation tiedot päivitetään operaatiolistaan. Tällainen hae-ja-käsittele -tyylinen menettely on tyypillistä, koska se on helppoa ja suoraviivaista, mutta sen testattavuus kärsii, koska eri vaiheita ei voida testata yksinään. Kuvassa 22 on refaktoroitu versio samasta metodista.

```
function calculateSingle(operation) {
    return getValue(operation.id)
        .then(result => {
            return setValue(operation, result.data.value)
        })
        .then(updateOperationInList)
        .catch(err => err);
}

function getValue(id) {
    return $http.get(domain + 'api/ws/rest/operations/' + id + '/value');
}

function setValue(operation, value) {
    operation.price = value;
    if (operation.type.mainType === 2 && operation.price > 0) {
        operation.price = operation.price * -1;
    }

    return operation;
}

function updateOperationInList(operation) {
    var index = _.findIndex(service.operationList, op => {
        return op.id === operation.id
    });
    service.operationList[index] = operation;

    return operation;
}
```

KUVA 22. Testattavan metodin calculateSingle refaktoroitu versio

Alkuperäisen metodin toiminnallisuus on säilynyt samana, mutta sen eri vaiheet ovat nyt helpommin testattavissa. Vaikkakaan kyseinen esimerkki ei ole monimutkaisimmasta päästä refaktoroinnin puolesta, se kuitenkin havainnollistaa, mitä sillä tavoitellaan. Huomattavaa on myös se, että vaikka koodin määrä kasvoi esimerkissä, sarjan viimeisin funktio updateOperationInList on toiminnallisuudeltaan varsin yleispätevä, minkä ansiosta sitä voidaan käyttää jatkossa kaikissa päivitystilanteissa. Eli loppujen lopuksi pitkällä aikavälillä syntyy vähemmän toistoa, ja lisäksi on mahdollista käyttää funktion kuvaavaa nimeä hyödyksi.

Näitä yksiköitä testatakseen, varsinaiset testit laaditaan erilliseen tiedostoon. Se mihin kyseinen tiedosto sijoitetaan, ja kuinka se nimetään, on laajalti mielipidekysymys. Tyypillinen ratkaisu - jota myös päädyin itse käyttämään - on luoda testattavan tiedoston

rinnalle toinen saman niminen tiedosto päätteellä ”.spec.js”. Tällöin testi on helposti löydettävissä sekä testinsuorittajalle, että kehittäjälle. Säännöllisen nimeämistyylin kautta testinsuorittaja osaa etsiä juuri kyseisellä päätteellä olevat tiedostot projektin kansioista.

```
1  describe('Testing OperationService', function() {
2      // Alustetaan testissä käytettävät muuttujat
3      var OperationService, operation;
4
5      // beforeEach on Jasminen tarjoama funktio, joka suoritetaan ennen jokaista testiä
6      beforeEach(function() {
7          // Alustetaan testattavan sovelluksen päämoduuli, joka on nimeltään 'app'
8          module('app');
9          // Määritellään polku kansioon, josta voidaan hakea oikeanmuotoista testidataa
10         jasmine.getJSONFixtures().fixturesPath = 'base/test/test-data';
11     });
12
13     // Injektoidaan testattava tehdasfunktio OperationService Angularin ngMock-moduulia hyödyntäen
14     // Nimen ympärillä on alaviivat siksi, että testeissä olisi mahdollista käyttää samaa nimeä ilman konfliktia
15     beforeEach(inject(function(_OperationService_) {
16         // Otetaan syötetty OperationService talteen muuttujaan
17         OperationService = _OperationService_;
18         // Haetaan esimerkkidata erillisestä JSON-tiedostosta
19         operation = getJSONFixture('operation.json');
20     }));
21 });
```

KUVA 23. Tehdasfunktion alustus yksikkötestissä

Angular on luonut testaamisen tueksi erillisen kirjaston nimeltä ngMock, joka tarjoaa keskeiset metodit testattavan Angular-sovelluksen alustamiseksi yksikkötesteihin. Kun kyseinen kirjasto ladataan testinsuorittajaan, kehittäjällä on käytettävissä Angular-moduulin luonnissa käytettävä module-metodi (kuva 23, rivi 8), sekä riippuvuuksien injektoinnissa käytettävä inject-metodi (rivi 15). Näitä kahta, sekä Jasminen tarjoamaa beforeEach-funktiota hyödyntämällä voidaan varmistaa, että testattava moduuli ja sen halutut ominaisuudet ovat käytettävissä kaikissa testeissä. Käytän testeissä oikeanmallista operaatio-objektia, joka sen pituuden vuoksi olisi epäkäytännöllistä säilöä testitiedostossa suoraan. Tästä syystä haen kyseisen datan erillisestä JSON-tiedostosta Jasminen getJSONFixtures-metodilla. Tällainen modulaarinen lähestymistapa myös helpottaa testien ylläpitoa, koska tarvittavat muutokset operaatio-objektin rakenteessa tulee jatkossa tehdä vain yhteen tiedostoon.

```
21
22      describe("Testing setValue function", function() {
23          it('should set a positive value to the operation object', function() {
24              // Varmistetaan, että operaation mainType on 1
25              operation.type.mainType = 1;
26              // Kutsutaan testattaa funktiota ja asetetaan sen paluuarvo operaatioon
27              operation = OperationService.setPrice(operation, 200);
28              // Odotetaan operaation hinnan olevan positiivinen
29              expect(operation.price).toEqual(200);
30          });
31
32          it('should set a negative value to the operation object', function() {
33              operation.type.mainType = 2;
34              operation = OperationService.setPrice(operation, 200);
35              // Koska mainType on 2, hinnan odotetaan olevan negatiivinen
36              expect(operation.price).toEqual(-200);
37          });
38      });
39  });
```

KUVA 24. Tehdasfunktion setValue -metodin testaus

Pitääkseni esittelyn lyhyempänä käyn testattavasta calculateSingle-funktiosta läpi vain sen yhden osan, setValue-funktion testauksen (kuva 24). Koska sen tehtävä on käytännössä sitoa haettu hinta joko positiivisena tai negatiivisena operaatio-objektiin, tämä on vähintä mitä testissä halutaan varmistaa. Laadin tätä varten kaksi testitapausta, joissa funktiota kutsutaan samoilla arvoilla, ainoana poikkeuksena operaatio-objektin mainType-ominaisuuden arvo.

4.2.2 Käsittelijät (controller)

Käsittelijän tehtävä on hallita näkymän tilaa, sekä toimia välikätenä näkymän ja mallin välillä. Tämä tarkoittaa, että käsittelijä ei ideaalitilanteessa vastaa mallien eikä liiketoimintalogiikan luonnista, vaan se kutsuu niitä tarpeen mukaan erillisestä tehdasfunktiosta. Tällainen rakennejako selkeyttää tiedonkulkua sovelluksessa, ja sen kautta myös tiedostokoot pysyvät todennäköisemmin järkevän kokoisina, kun koodi on jaettu useampaan eri tiedostoon.

Tässä luvussa testauksen alla on käsittelijä nimeltä EstateOperationsController, joka kulkee käsi kädessä edellisessä luvussa testaamani tehdasfunktion kanssa. Käsittelijä vastaanottaa kyseiseltä tehdasfunktiolta käsiteltävän datan (mallin), jonka se voi sitoa näkymään. Tämä yhteys käsittelijän ja tehdasfunktion välillä on siis varsin keskeinen asia sovelluksen toimivuuden kannalta, joten se on hyvä varmistaa yksikkötesteissä.

Kuva 25 sisältää selkeyttämisen nimissä karsitun version testattavasta käsittelijästä ja edellä mainitusta yhteydestä.

```
1  angular.module('app.controllers')
2          .controller('EstateOperationsController', EstateOperationsController);
3
4  EstateOperationsController.$inject = ["$scope", "OperationService"];
5  function EstateOperationsController($scope, OperationService) {
6
7      $scope.operationList = OperationService.operationList;
8
9  }
```

KUVA 25. Mallin hakeminen tehdasfunktiosta käsittelijään

Tätä testatakseni loin jälleen testattavan tiedoston rinnalle toisen samannimisen tiedoston, päätteellä ”.spec.js”. Käsittelijän alustus poikkeaa tehdasfunktiosta siinä, että se joudutaan luomaan Angularin controller-rakentajametodia hyödyntäen. Kyseinen metodi on saatavilla $controller-palvelun (service) kautta, joka voidaan injektoida funktioon samalla tyylillä, kuin muutkin käsittelijän riippuvuudet. Ennen tätä tulee kuitenkin tehdä tarvittavat muuttuja- ja moduulialustukset (kuva 26, rivit 1–9), kuten tehdasfunktiotakin testattaessa.

```
1  describe('Testing EstateOperationsController', function() {
2
3      var $scope, OperationService;
4
5      beforeEach(function() {
6          module('app');
7          jasmine.getJSONFixtures().fixturesPath = 'base/test/test-data';
8      });
9
10     beforeEach(inject(function(_$rootScope_, _$controller_, _OperationService_) {
11         // Luodaan uusi skooppi $rootScope-objektin $new -metodilla
12         $scope = _$rootScope_.$new();
13         // Poimitaan OperationService muuttujaan
14         OperationService = _OperationService_;
15         // Haetaan erillisestä JSON-tiedostosta lista operaatioita, ja sidotaan se OperationServiceen
16         OperationService.operationList = getJSONFixture('operationList.json');
17
18         // Luodaan testattava käsittelijä $controller-palvelun avulla. Kutsuu käytännössä angular.controller -metodia
19         // Toisena parametrinä annetaan objekti niistä käsittelijän riippuvuuksista, joita käytetään testeissä
20         // Kolmas parametri on vaihtoehtoinen objekti, jonka avulla voidaan alustaa ylimääräisiä muuttujia skooppiin
21         _$controller_(
22             'EstateOperationsController',
23             { $scope: $scope, OperationService: OperationService },
24             {}
25         );
26     }));
27 });
```

KUVA 26. Käsittelijän alustus yksikkötestissä

Alustusten jälkeen varsinainen käsittelijän luominen voi alkaa. Jotta halutun käsittelijän ominaisuudet olisivat käytettävissä testissä, ne tulee alustaa etukäteen ja syöttää käsittelijälle sen luontivaiheessa. Tätä kautta käsittelijä ei itse yritä alustaa kyseisiä muuttujia, vaan se tyytyy käyttämään testissä alustettuja objekteja sen sijaan. Muilta osin käsittelijä toimii aivan kuten normaalistikin, minkä seurauksesta sen käyttöä voidaan simuloida testitiedostosta käsin.

Testissä alustetut muuttujat toimivat suorana viittauksena niihin objekteihin, jotka sidotaan käsittelijään sen luonnin yhteydessä. Tällä tiedolla voidaankin siis olettaa, että tehdasfunktioon sidottu operaatiolista (kuva 26, rivi 16) on käytettävissä myös käsittelijän skoopin kautta, kuten määrittelin aiemmin kuvassa 25. Tämä yhteys voidaan varmistaa kuvan 27 esittämän testin kautta.

```
22
23      it('should have have $scope.operationList defined as a reference to OperationService.operationList', function() {
24          expect($scope.operationList).toBe(OperationService.operationList);
25          expect($scope.operationList.length).toEqual(174);
26      });
27  });
```

KUVA 27. Testi, joka vahvistaa skoopin sisältämän operaatiolistan olevan suora viittaus tehdasfunktion sisältämään listaan

Testissä käytetty Jasminen toBe-metodi vertailee syötettyjä listoja, ja odottaa niiden olevan viittaus tismalleen samaan listaan. Tästä syystä kyseinen testi epäonnistuisi, jos hakisin skoopin sisältämään operaatiolistaan erillisen kopion samasta listasta. Vaikka ne olisivat sisällöltään samoja, ne eivät kuitenkaan ole käytännössä yksi ja sama objekti. Koska tiedän listan sisältävän 174 objektia, voin vahvistaa samalla myös sen käyttämällä toista Jasminen vertailumetodia toEqual, joka on muuten sama kuin edellinen toBe-metodi, mutta se vertailee annettuja arvoja vain pinnallisesti.

4.2.3 Komponentit (directive, component)

Angularin komponentit ovat yksinkertainen keino luoda uudelleenkäytettäviä elementtejä sivustolle. Ne koostuvat html-mallista ja käsittelijästä siinä missä muukin Angular-sisältö, mutta kyseiset määritykset on eristetty varsinaisesta käyttökontekstista. Sen sijaan, että monimutkaista html-syntaksia kopioitaisiin paikasta toiseen manuaalisesti,

Angular vastaa sen luonnista sivunlatauksen yhteydessä. Kehittäjän näkökulmasta komponentin käyttö tapahtuu vain yksittäisellä html-elementillä, jonka kautta komponentti vastaanottaa myös käsiteltävän datan, käyttäen html-attribuutteja. Käytännössä ympäröivällä kontekstilla ei siis ole merkitystä, kunhan komponentin käyttämä tieto tulee välitetyksi ulkopuolelta.

```
1  angular
2      .module('app.directives')
3      // Komponentin luominen angular.directive -rakentajametodilla
4      .directive('mhgList', mhgList)
5      // Komponentin käsittelijän luominen
6      .controller('ListCtrl', ListCtrl);
7
8  function mhgList() {
9      return {
10         // Käyttö tapahtuu pelkästään html-elementin kautta
11         restrict: 'E',
12         // Komponentin html-malli
13         templateUrl: 'app/components/list/mhgList.html',
14         // Tiedot, jotka sidotaan komponentin skooppiin html-attribuuttien kautta
15         scope: {
16             repeat: '=',
17             options: '='
18         },
19         // Komponentin käsittelijän nimi
20         controller: "ListCtrl"
21     };
22 }
```

KUVA 28. Komponentin määritykset

Testattava komponentti on sovelluksessa varsin runsaassa käytössä oleva mhgList, joka on käytännössä taulukkomainen listaelementti. Sen sisältö määräytyy sille syötettävän JavaScript-listan, sekä erillisen asetusobjektin kautta. Kuvassa 28 on karsittu versio kyseisestä komponentista ja sen määrityksistä. Kun Angular havaitsee html-koodista <mhg-list>-elementin, se osaa luoda lopullisen elementin näiden määritysten mukaisesti.

```
24 ListCtrl.$inject = ["$scope", "MiscService"];
25 function ListCtrl($scope, MiscService) {
26
27     $scope.$watch("repeat", function(newV, oldV) {
28         $scope.filteredList = MiscService.filterList($scope.repeat, $scope.options);
29     }, true);
30
31 }
```

KUVA 29. Komponentin käsittelijä, ja testattava ominaisuus

Varsinainen testattava ominaisuus löytyy komponentin käsittelijästä, joka on määriteltynä kuvassa 29. Kyseessä on skoopin $watch-metodi, jonka tehtävä on tarkkailla haluttua arvoa muutosten varalta. Tässä tapauksessa se tarkkailee repeat-listaa, joka välitetään sille komponentin alustuksessa. Jos jokin listan arvoista muuttuu lainkaan, metodin toisena parametrinä syötetty funktio suoritetaan. Tällä funktiolla on käytettävissään sekä listan edellinen, että uusi versio, joita vertailemalla muutoksia on mahdollista tutkia tarkemmin. Tässä tapauksessa ainoa suoritettava toimenpide on kyseisen listan suodattaminen filterList-metodilla, joka on määritetty omatekemässä MiscService-tehdasfunktiossa.

```
describe('Testing the mhg-list directive', function() {
    var $scope, parentScope, MiscService;

    beforeEach(module('app'));

    beforeEach(inject(function(_$rootScope_, _$controller_, _$compile_, _MiscService_) {
        MiscService = _MiscService_;

        // Luodaan uusi skooppi, ja sidotaan siihen arvot, jotka halutaan välittää komponentille
        // Arvot voidaan myöhemmin määrittää testikohtaisesti, riippuen siitä mitä halutaan testata
        parentScope = _$rootScope_.$new();
        parentScope.someArray = [];
        parentScope.someOptions = {};

        // Luodaan uusi Angular-elementti komponentin html-syntaksia käyttäen, ja sidotaan skoopin arvot siihen
        var element = angular.element('<mhg-list repeat="someArray" options="someOptions"></mhg-list>');
        // $compile-palvelu tunnistaa elementin olevan määrittelemämme komponentti, ja luo sen annetun skoopin sisälle
        _$compile_(element)(parentScope);
        // Komponentti luodaan vasta Angularin digest-syklin aikana, joka voidaan laukaista manuaalisesti $digest -metodilla
        parentScope.$digest();
        // Kun komponentti on luotu, sen skooppiin päästään käsiksi kutsumalla elementin isolateScope -metodia
        $scope = element.isolateScope();
    }));
});
```

KUVA 30. Komponentin yksikkötestien alustus

Koska komponentti vastaanottaa käsiteltävän datan ulkopuolelta, sitä tulee kyetä simuloimaan myös yksikkötesteissä. Tämä monimutkaistaa komponentin alustusta, koska pelkkä testattavan komponentin luominen ei riitä, vaan myös komponenttia ympäröivä skooppi tulee luoda ennen komponentin luomista. Kyseiseen skooppiin sidotut arvot ovat tämän jälkeen välitettävissä komponentille kuvan 30 määritysten mukaisesti.

```
25      describe('Testing the $scope.repeat watcher', function() {
26          it('should call the filterList method in MiscService when $scope.repeat changes', function() {
27              // Luodaan oikean filterList -metodin tilalle tynkä, joka vakoilee kutsutaanko sitä testin suorituksen aikana
28              MiscService.filterList = jasmine.createSpy();
29              // Lisätään satunnainen objekti alkuperäiseen listaan, joka on komponentin ulkopuolella
30              parentScope.someArray.push({id: 1, number: 1});
31              // Funktion ei odoteta vielä tulleen kutsutuksi, koska digest-sykli ei ole laukaistu
32              expect(MiscService.filterList).not.toHaveBeenCalled();
33              // Syklin laukaistessa $watcher -metodin voidaan odottaa kutsuneen filterList -metodia, koska lista on muuttunut
34              parentScope.$digest();
35              expect(MiscService.filterList).toHaveBeenCalled();
36          });
37      });
38  });
```

KUVA 31. Komponentin skoopissa olevan $watcher -metodin testaus

Kuvassa 31 luotu yksikkötesti varmistaa, että komponentin sisällä määritelty tarkkailijafunktio tulee suoritetuksi, jos sen tarkkailema lista muuttuu komponentin ulkopuolella. Huomioitavaa tässä testissä on se, että funktion kutsuma filterList -metodi ylikirjoitetaan testin alussa Jasminen ns. vakoilijafunktiolla, jonka tarkoituksena on yksinkertaisesti seurata, tuleeko se kutsutuksi testin aikana. Yksikkötestauksen näkökulmasta tämä onkin ainoa asia, jota kyseisessä testissä halutaan tarkistaa: tuleeko filterList kutsutuksi oikeaan aikaan? Jos alkuperäisen filterListin lopputulosta haluttaisiin testata, se tulisi tehdä sen omassa yksikkötestissä.

5 PÄÄTÄNTÖ

Työn tavoitteena oli laatia Metsäpaikan sovelluskehitykseen toimiva testausympäristö, jonka avulla sovelluksen toimintaan liittyviä ongelmia olisi mahdollista havaita aikaisessa vaiheessa. Tällä määritelmällä voin todeta onnistuneeni tavoitteen saavuttamisessa, sillä testaustyökalut toden totta tuli integroitua projektiin ja ne ovat nyt kehitystiimin käytettävissä.

Työn aiheeseen liittyy kuitenkin ongelmia, joista suurin osa juontaa juurensa Angularista ja sen kaltaisista sovelluskehyksistä yleisesti. Koska vastaavien työkalujen tarkoitus on tarjota kaiken kattava kokonaisuus käyttöliittymien luomiseen, ne joutuvat tekemään paljon asioita ns. pellin alla. Vastaava abstrahointi vähentää kehittäjän taakkaa monella tasolla, mutta se aiheuttaa haasteita testauksen näkökulmasta, sillä kyseiset pellin alla tapahtuvat asiat tulee kyetä ottaa huomioon testejä kirjoittaessa. Tämä nostaa kynnystä testauksen aloittamiseksi, minkä takia se helposti jätetään tekemättä täysin.

Koen opinnäytetyöni lieventävän tätä aloittamisen tuskaa Angular-sovellusten osalta, vaikka se käytännössä vain raapaisee pintaa testattavien ominaisuuksien suhteen. Työssä käyttämäni esimerkit keskittyivätkin enemmän siihen, kuinka keskeisten Angular-ominaisuuksien testauksessa päästään alkuun, ei niinkään siihen, kuinka varsinaisia yksiköitä testataan. Jälkimmäinen asia onkin sellainen osa-alue, jossa työtä olisi vielä mahdollista kehittää eteenpäin. Yksinkertaisuudestaan huolimatta testit tuovat kuitenkin esille tyypillisiä tilanteita, joita yksikkötesteissä halutaan ottaa huomioon.

Toinen, erityisesti JavaScript-sovelluskehyksille ominainen ongelma on niiden usein varsin lyhyt elinkaari. Uusien tekniikoiden kehittyessä vanhat sovelluskehykset tulevat herkästi sivuutetuiksi, minkä vuoksi tämänkin opinnäytetyön kaltaiset ”oppaat” ovat relevantteja korkeintaan muutaman vuoden ajan. Myös Angular on juuri tällaisen vaihdoksen partaalla, sillä sen seuraava versio, Angular 2 on täysi uudelleenkirjoitus työssä käytettyyn versioon (1.5) verrattuna, eikä selkeää päivitysväylää näiden versioiden välillä vielä kirjoittamisen hetkellä ole olemassa. Tästä johtuen vanhemmatkin versiot tulevat odotettavasti pysymään kuvioissa vielä ainakin pari vuotta, kun yritykset etsivät keinoja palveluidensa päivittämiseksi.

Metsäpaikan jatkokehitystä ajatellen puitteet yksikkötestauksen suorittamiselle ovat hyvät. Yksikkötestaus on kuitenkin vain yksi osa ohjelmistotestausta, minkä vuoksi suurimmat mahdollisuudet testauksen parantamiselle löytyvät muista testausmuodoista. Seuraava luonnollinen askel kohti parempaa testikattavuutta olisi varmistaa yksiköiden keskinäinen toiminta integraatiotestauksen kautta. Olin suunnitellut tämän olevan osa opinnäytetyötäni yksikkötestauksen rinnalla, mutta Angularin tuomat lisähaasteet siirsivät tämän toteuttamisen myöhemmälle.

Kaiken kaikkiaan opinnäytetyö oli kokemuksena varsin positiivinen. Sen kautta sain hyvän syyn perehtyä sovellusten testaamiseen, joka on aihealueena jäänyt huomioimatta tähänastisten opintojen ja töiden aikana. Se toi myös uusia näkökulmia laadukkaan koodin laatimiseen, mikä on itselleni henkilökohtaisesti tärkeä asia kasvavana kehittäjänä. Toimeksiannon kautta sain myös syvennettyä aikaisemmin kerryttämääni ymmärrystä Angularista ja sen toimintaperiaatteista, mikä tulee olemaan hyödyksi lähitulevaisuuden kehitysprojekteissa.

LÄHTEET

Austin, Andrew 2014. An Overview of AngularJS for Managers. http://andrewaustin.com/an-overview-of-angularjs-for-managers/. Blogi. Päivitetty 27.8.2014. Luettu 6.2.2016.

Jasmine 2015. Introduction. WWW-dokumentti. http://jasmine.github.io/2.4/introduction.html. Ei päivitystietoja. Luettu 20.2.2016.

Karma 2016. WWW-dokumentti. https://karma-runner.github.io/0.13/index.html. Ei päivitystietoja. Luettu 21.2.2016.

Karma 2014. How It Works. WWW-dokumentti. http://karma-runner.github.io/0.13/intro/how-it-works.html. Päivitetty 24.3.2014. Luettu 22.2.2016.

Kasurinen, Jussi 2016. Ohjelmistotestauksen perusteet. Lappeenrannan teknillinen yliopisto. PDF-dokumentti. https://noppa.lut.fi/noppa/opintojakso/ct60a4150/luennot/luennon_4_kalvot.pdf. Päivitetty 1.2.2016. Luettu 23.3.2016.

Korjus, Nina 2015. Testauksen perusteet. Lappeenrannan teknillinen yliopisto. PDF-dokumentti. https://noppa.lut.fi/noppa/opintojakso/ct60a4150/materiaali/ohjelmistotestauksen_perusteet_-manuaali.pdf. Päivitetty 11.3.2015. Luettu 13.3.2016.

Lyytinen, Timo 2013. HTML5 web-sovellukset. https://www.vincit.fi/blog/html5-web-sovellukset/. Blogi. Päivitetty 5.6.2013. Luettu 6.2.2016.

Miško Hevery and Brad Green - Keynote - NG-Conf 2014. Videoklippi. https://www.youtube.com/watch?v=r1A1VR0ibIQ. Päivitetty 16.1.2014. Katsottu 6.2.2015

MSDN 2016. Unit Testing. WWW-dokumentti. https://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx. Päivitetty 4.4.2016. Luettu 7.4.2016.

Mozilla Developer Network 2015. MVC architecture. Artikkeli. https://developer.mozilla.org/en-US/Apps/Build/Modern_web_app_architecture/MVC_architecture. Päivitetty 14.3.2015. Luettu 13.2.2016.

Mozilla Developer Network 2016. XMLHHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest. Artikkeli. Päivitetty 4.2.2016. Luettu 9.2.2016.

Myers, Glenford J. 1979. The Art of Software Testing. New York, USA: John Wily & Sons Inc.

Papa, John 2013. SPA and the Single Page Myth. http://www.johnpapa.net/pageinspa/. Blogi. Artikkeli. Päivitetty 30.11.2013. Luettu 7.2.2016.

Ruotsalainen, Teemu 2014. Perinnekoodin refaktorointi ja testattavuus. Tampereen yliopisto. Pro gradu -tutkielma. PDF-dokumentti. Päivitetty 21.5.2014. Luettu 3.5.2016.

Sainio, Laura 2010. Ohjelmistotestauksen menetelmät ja työvälineet. PDF-dokumentti. https://www.theseus.fi/bitstream/handle/10024/12297/Sainio_Laura_liite1.pdf. Päivitetty 15.3.2010. Luettu 21.4.2016.

Shimanovsky, Serge 2015. Multi page web applications vs. single page web applications. http://www.eikospartners.com/blog/multi-page-web-applications-vs.-single-page-web-applications. Blogi. Artikkeli. Päivitetty 9.7.2015. Luettu 7.2.2016.

Tietojenkäsittelytieteen laitos 2009. Ohjelmistoprosessit ja ohjelmistojen laatu. PDF-dokumentti. https://www.cs.helsinki.fi/u/taina/opol/k-2009/pdf/luku-6_2.pdf. Päivitetty 23.9.2009. Luettu 25.3.2016.

Tietojenkäsittelytieteen laitos 2011. Yksikkötestaamisesta. PDF-dokumentti. http://www.cs.helsinki.fi/u/avihavai/edutainment/2011/ohma/7-yksikkotestaamisesta.pdf. Päivitetty 9.8.2011. Luettu 1.2.2016.

Tolvanen, Perttu 2013. Ketteryys haltuun: Ketterän kehityksen yleiset periaatteet. http://www.meteoriitti.com/2013/06/06/ketteryys-haltuun-ketteran-kehityksen-yleiset-periaatteet/. Päivitetty 6.6.2013. Luettu 25.3.2016.

Tuovinen, Antti-Pekka 2013. Ohjelmistotestauksen perusteita I. PDF-dokumentti. https://www.cs.helsinki.fi/u/aptuovin/testaus/Ohj_testaus_2013_1.pdf. Päivitetty 12.3.2013. Luettu 8.3.2016.

Viskari, Mikko 2015. SPA-sovellukset. http://www.eatech.fi/fi/blogi/spa-sovellukset. Blogi. Artikkeli. Päivitetty 18.11.2015. Luettu 6.2.2016.

XMLHttpRequest advanced features 2016. Can I use. WWW-dokumentti. http://caniuse.com/#feat=xhr2. Ei päivitystietoja. Luettu 9.2.2016.

Zavelevsky, Doron 2014. Protractor vs. Selenium: Which is Easier? WWW-dokumentti. Artikkeli. http://testautomation.applitools.com/post/94994807787/protractor-vs-selenium-which-is-easier. Päivitetty 17.8.2014. Luettu 1.3.2016.