VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Oskari Eskelinen

JAVASCRIPT-KOODIN MODERNISOINTI

Case Visma InCommunity Oy

Liiketalous
2017

VAASAN AMMATTIKORKEAKOULU
Tietojenkäsittely

TIIVISTELMÄ

Tekijä	Oskari Eskelinen
Opinnäytetyön nimi	JavaScript-koodin modernisointi
Vuosi	2017
Kieli	suomi
Sivumäärä	44
Ohjaaja	Sirkka Hellman

Tämä opinnäytetyö käsittelee vanhan JavaScript-koodikirjaston modernisointia. Työn teoriaosassa tutkitaan, miten vanhaa JavaScript-koodia voidaan modernisoida ja mitä hyötyjä modernisoinnista saadaan. Työn toimeksiantajana toimi vaasalainen Visma InCommunity Oy.

Opinnäytetyössä selvitetään koodin alkuperäisen tilan suurimmat puutteet ja teknologiat, joita näiden puutteiden kehittämiseen tarvitaan. Tärkeimpinä kehityskohteina työssä käsitellään koodin modulaarisuutta, yksikkötestien kattavuutta ja modernien ECMAScript-ominaisuuksien hyödyntämistä. Lisäksi työhön kuuluu tekninen toteutusosa, joka sisältää muun muassa moduulien rakentamisen ja yksikkötestien kirjoittamisen.

Työn tuloksena alkuperäiset tavoitteet on saatu pääosin täytettyä. Alkuperäiseen tilaan verrattuna koodin laatu on kasvanut. Toteutuksessa on saavutettu toimiva moduulirakenne, yksikkötestien kattavuus on saatu hyvälle tasolle ja ECMAScript 2015:n ominaisuuksia on hyödynnetty. Työn tulokset jäävät toimeksiantajayrityksessä tuotantokäyttöön ja lisäksi työn tekijän asiantuntemus aihepiiristä on kasvanut merkittävästi.

Avainsanat ohjelmointi, JavaScript, ECMAScript, yksikkötestaus

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietojenkäsittely

ABSTRACT

Author	Oskari Eskelinen
Title	Modernising JavaScript Code
Year	2017
Language	Finnish
Pages	44
Name of Supervisor	Sirkka Hellman

This thesis studied the modernisation of an existing JavaScript code library. The theoretical section of the thesis studies the ways old JavaScript code can be modernised and the benefits modernisation offers. The client of this thesis was a Vaasa-based company, Visma InCommunity Oy.

The main weaknesses and issues of the original code are covered in the thesis, as well as the tools and the technologies needed to rectify the identified issues. The key areas covered are code modularity, unit test coverage and modern ECMAScript features. The actual development work is also covered in the thesis, including building a module system and writing unit tests.

The goals set at the very beginning were mostly met. Compared to its original state, the quality of the code improved. A fully functional modular system was built, a high unit test coverage was achieved, and ECMAScript 2015 features were taken advantage of. The results will be integrated into the main product and used in production, and the thesis writer's expertise on the subject has grown tremendously.

Keywords programming, JavaScript, ECMAScript, unit testing

SISÄLLYS

TIIVISTELMÄ
ABSTRACT
KÄSITTEET

1 JOHDANTO 9
1.1 Toimeksiantaja 9
1.2 Aiheen tausta 9
1.3 Tavoitteet 10
2 UUDISTUSKOHTEIDEN KARTOITUS 11
2.1 Taustatietoja 11
2.2 Modulaarisuus 12
2.3 Yksikkötestattavuus 13
2.4 ECMAScript 2015 14
2.4.1 Nuolifunktiot 14
2.4.2 Let ja const 15
2.4.3 Luokat 17
2.4.4 Moduulit 18
3 KÄYTETYT TEKNOLOGIAT 20
3.1 Node.js 20
3.2 Jasmine 21
3.3 Grunt 22
3.4 Karma 23
3.5 Babel 25
3.6 Webpack 26
4 TOTEUTUS 27
4.1 Suunnittelu 27
4.2 Ohjelmointi 31
4.2.1 getLokalisointiFromElement 31
4.2.2 jsDateToDelphiDateTime 34
4.2.3 datetimeIsInPast 36
4.3 Yhteensopivuus 38

4.3.1 JavaScriptEndpoint 38
4.3.2 Hakijafunktiot 39
4.3.3 Viittausten korjaus 39
5 YHTEENVETO 41
LÄHTEET 43

KÄSITTEET

ECMAScript

Ecma Internationalin standardoima määritys, johon mm. JavaScript perustuu.

JavaScript

ECMAScript-standardiin perustuva ohjelmointikieli.

Node.js

Googlen kehittämään V8-JavaScript-moottoriin perustuva ajoympäristö.

Electron

Node.js:ään pohjautuva sovelluskehys, jonka avulla voidaan kirjoittaa työpöytäsovelluksia käyttäen JavaScriptiä.

DOM

Lyhenne sanoista Document Object Model. Tarkoittaa tapaa kuvata erilaisten dokumenttien rakennetta puuna.

LESS

Kielilaajennus CSS-kieleen. LESS-koodi käännetään kääntäjän avulla takaisin CSS:ksi.

CommonJS

Työryhmä, jonka tehtävä on laajentaa JavaScript-ekosysteemiä selaimen ulkopuolelle.

Delphi

Object Pascal -kieleen perustuva graafinen ohjelmointiympäristö.

KUVIO- JA TAULUKKOLUETTELO

Kuvio 1. Globaali funktio addevent.js:ssä 12
Kuvio 2. Globaalin funktion näkyvyys selaimessa 12
Kuvio 3. Nuolifunktion kolmelta riviltä yhdelle riville tiivistämä toiminto 15
Kuvio 4. Var-muuttujan toiminta 16
Kuvio 5. Let-muuttujan toiminta 16
Kuvio 6. Const-muuttujan toiminta 17
Kuvio 7. Luokan määrittely ja käyttäminen 18
Kuvio 8. Funktion vienti 19
Kuvio 9. Funktion tuonti 19
Kuvio 10. Kokonainen Jasmine-suite kahdella testillä 21
Kuvio 11. Wilman Gruntfilessä määritetyt tehtävät 22
Kuvio 12. Karman ajaminen konsoli-ikkunasta 24
Kuvio 13. Yksikkötestien kattavuusraportti 24
Kuvio 14. Babel-käännös ECMAScript 2015:stä ECMAScript 5:ksi 25
Kuvio 15. Webpackin ajo kehitystilassa Gruntin avulla 26
Kuvio 16. Moduulikokonaisuuden kansiorakenne 28
Kuvio 17. Yksittäisen moduulin rakenne 29
Kuvio 18. string-moduulin index.js 29
Kuvio 19. Helpers-kansion index.js 30
Kuvio 20. Helpers-moduulin käyttö 30
Kuvio 21. getLokalisointiFromElement, alkuperäinen koodi 31
Kuvio 22. getLokalisointiFromElement, testien pohja 32
Kuvio 23. getLokalisointiFromElement, valmiit yksikkötestit 33
Kuvio 24. getLokalisointiFromElement, onnistunut testien ajo 33
Kuvio 25. getLokalisointiFromElement, lopullinen koodi 34
Kuvio 26. jsDateToDelphiDateTime, alkuperäinen koodi 35
Kuvio 27. jsDateToDelphiDateTime, valmis yksikkötesti 35
Kuvio 28. jsDateToDelphiDateTime, lopullinen koodi 36
Kuvio 29. datetimeIsInPast, alkuperäinen koodi 36
Kuvio 30. datetimeIsInPast, valmiit yksikkötestit 37
Kuvio 31. datetimeIsInPast, lopullinen koodi 37

Kuvio 32. SetJavaScriptEndpoint-metodi 38
Kuvio 33. Helpers-hakija 39
Kuvio 34. Vanhentunut funktioviittaus 39
Kuvio 35. Päivitetty funktioviittaus 40
Kuvio 36. Yksikkötestien kattavuus työn loppuvaiheessa 42

Taulukko 1. Grunt-tehtävät 23
Taulukko 2. Moduulien tehtävänjako 27

1 JOHDANTO

Tämä opinnäytetyö käsittelee toimeksiantajan vanhan, tuotantokäytössä olevan JavaScript-koodin modernisointia usealla eri osa-alueella. Modernisointityö keskitetään toimeksiantajan Wilma-tuotteen yksittäisen kirjaston koodiin.

Työn alussa käydään läpi JavaScriptin historiaa lyhyesti sekä analysoidaan modernisoitavan kirjaston lähtötilannetta, sen suurimpia puutteita ja tärkeimpiä uudistustarpeita. Lisäksi työssä käydään läpi löydettyjen uudistustarpeiden toteuttamiseen tarvittavat teknologiat ja varsinainen tekninen toteutus.

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimi Visma InCommunity Oy. Yritys on erikoistunut koulu- ja oppilaitoshallintoon ja se on alansa markkinajohtaja Suomessa. Yritys työllistää noin viisikymmentä henkilöä ja sen toimipiste sijaitsee Vaasan keskustassa. Yritys on osa pohjoismaista Visma-konsernia.

Visma InCommunity Oy:n tuoteperheeseen kuuluu:

- **Primus**-tietokantaohjelma, joka toimii oppilaitoshallinnon kokonaisjärjestelmän ytimenä (Visma InSchool 2017)
- **Kurre**-työjärjestysohjelma, jonka avulla mm. suunnitellaan lukuvuoden työjärjestykset ja hallitaan opetusryhmiä (Visma InSchool 2017)
- **Wilma**, joka on Primuksen ja Kurren yhteinen www-käyttöliittymä (Visma InSchool 2017)
    - Käyttäjiä ovat mm. huoltajat, opiskelijat ja opettajat

1.2 Aiheen tausta

Opinnäytetyön tekijä työskentelee yrityksessä ohjelmistosuunnittelijana ja idea opinnäytetyöhön syntyi normaalin tuotekehityksen ohessa. Eräs laajalti Wilmassa käytetty JavaScript-kirjasto, *addevent.js*, oli vuosien varrella paisunut useiden tuhansien rivien mittaiseksi ja monella osa-alueella laadullisesti heikoksi muuhun

ohjelman koodiin verrattuna. Opinnäytetyössä keskitytään tämän kirjaston modernisointiin.

Lähtötilanteessa kirjasto on laaja, kaikenkattava paketti, joka sisältää erilaisia funktioita Wilman kehityksen tueksi. Nämä funktiot käsittelevät muun muassa aikaa, merkkijonoja ja lomakkeita. Kuten kirjaston nimestäkin voi päätellä, alun perin sen tarkoitus oli tarjota Wilman kehittäjille työkaluja erilaisten tapahtumien hallintaan.

Työn tärkeimpiä tutkimuskysymyksiä ovat:

- Mitä kaikkea koodissa olisi tarpeen uudistaa?
- Mitä hyötyjä koodin uudistamisesta saadaan?

1.3 Tavoitteet

Modernisoitavan kirjaston tärkeitä uudistuskohteita ovat muun muassa uudemman ECMAScript-standardin ominaisuuksien hyödyntäminen, yksikkötestien kattavuuden kasvattaminen ja modulaarisemman rakenteen saavuttaminen. Näiden uudistuskohteiden lisäksi työn tavoitteena oli vähentää vuosien varrella kertynyttä teknistä velkaa sekä lisätä koodin selkeyttä ja helppokäyttöisyyttä kehittäjien näkökulmasta. Tavoitteena oli myös lisätä omaa asiantuntemusta moderneista webteknologioista, Node.js-alustasta ja testattavan koodin kirjoittamisesta.

2 UUDISTUSKOHTEIDEN KARTOITUS

Tässä kappaleessa käsitellään lyhyesti JavaScriptin historiaa, sen kehitystä vuosien varrella ja sitä, miksi työssä tehtävä modernisointi on tarpeellista. Lisäksi kappaleessa kartoitetaan työssä käsiteltävän *addevent.js*-kirjaston puutteita, kehitystarpeita ja sitä, miten näitä tarpeita tullaan kehittämään parempaan suuntaan.

2.1 Taustatietoja

JavaScript ja sen käyttötarkoitukset ovat muuttuneet valtavasti viimeisen kymmenen vuoden aikana. JavaScriptin tullessa markkinoille 1990-luvulla, sitä käytettiin pääsääntöisesti pienten ominaisuuksien ja yksinkertaisen interaktiivisuuden lisäämiseen verkkosivuille. Nykyään JavaScript on täysiverinen ohjelmointikieli, jota käytetään laajojen sovellusten toteuttamiseen.

JavaScriptin perinteisen selaimiin pohjautuvan käyttötarkoituksen lisäksi Node.js-alusta on valtavassa suosiossa palvelinpuolella. Myös työpöytäsovelluksia voi kirjoittaa täysin HTML-, CSS- ja JavaScript-kieliä hyödyntäen Electron-ohjelmistokehyksen päälle (Electron 2017). Vuonna 2017 ohjelmistokehityksen harrastajille ja ammattilaisille järjestetyssä kyselyssä jopa 62,5 % kaikista vastaajista ilmoitti käyttävänsä JavaScriptiä jossain muodossa ja se onkin jo viidettä vuotta peräkkäin maailman suosituin ohjelmointikieli (Stack Overflow 2017). Tämän nopean kehityksen ansiosta JavaScript-koodi myös vanhenee nopeasti.

Wilman ensimmäinen versio julkaistiin jo vuonna 2000 ja se kantaa mukanaan paljon vuosien varrella kertynyttä teknistä velkaa. Työssä käsiteltävä kirjasto, *addevent.js*, on vuosien varrella paisunut kaiken kattavaksi paketiksi, joka sisältää funktioita muun muassa ajan, merkkijonojen, tapahtumien ja lomakkeiden käsittelyyn. Myös osa kriittisestä toiminnallisuudesta kuuluu tämän kirjaston toiminnallisuuteen, kuten käyttäjän uloskirjautumisen hoitava koodi. Vain pieni osa kirjaston koodista liittyy sen alkuperäiseen tarkoitukseen. Kirjaston koodissa on useita puutteita ja korjauksen tarpeessa olevia osia, joita käsitellään seuraavaksi.

2.2 Modulaarisuus

JavaScriptin ajoympäristössä on aina olemassa *globaali objekti*. Kun JavaScript-koodia ajetaan selainympäristössä, *web worker* -taustatehtäviä lukuun ottamatta globaali objekti on aina *Window* (Mozilla Developer Network 2017). Valtaosa *addevent.js*:n koodista on kirjoitettu suoraan globaaliin nimiavaruuteen.

```
function getPrintLinkText() {
    switch (document.documentElement ? document.documentElement.lang : '') {
        case 'sv': return 'Skriv ut';
        case 'en': return 'Print';
        default: return 'Tulosta';
    }
}
```

Kuvio 1. Globaali funktio addevent.js:ssä

Kuvion 1 esimerkkifunktio, *getPrintLinkText*, on yksi sadoista globaaliin nimiavaruuteen kirjoitetuista funktioista, joita *addevent.js* sisältää. Sen tarkoitus on hakea käyttäjän selaimen kielen perusteella oikein lokalisoitu teksti tulostuspainikkeisiin.

```
» typeof window.getPrintLinkText
← "function"
```

Kuvio 2. Globaalin funktion näkyvyys selaimessa

Kuviossa 2 demonstroidaan kuvio 1:ssä esitellyn funktion näkyvyyttä selaimessa. Esimerkistä käy ilmi, että funktio sijaitsee suoraan globaalin *Window*-objektin alla. Wilman kaltaisissa laajoissa JavaScript-sovelluksissa olisi syytä välttää tämänkaltaista globaalin nimiavaruuden täyttämistä. Globaalin nimiavaruuden täyttäminen saattaa johtaa muun muassa muistinkäytön lisääntymiseen ja konflikteihin muuttujien ja funktioiden nimissä (Stack Overflow 2015). Konfliktit johtavat herkästi ennalta arvaamattomiin ja vaikeasti korjattaviin ongelmiin.

Koodin jakaminen moduuleihin edistää turvallisuuden lisäksi yksikkötestattavuutta, ylläpidettävyyttä ja helppokäyttöisyyttä (Preethi Kasireddy 2016). Tästä syystä työssä *addevent.js*:n sisältöä tullaan jakamaan pieniin, selkeästi rajattuihin moduu-

leihin; esimerkiksi merkkijono-, aika- ja lomakefunktiot jaetaan omiksi moduuleikseen, kukin omaan tiedostoonsa.

Modulaarisuutta voidaan JavaScriptin maailmassa toteuttaa monella tavalla. Ne ovat myös yksi ECMAScript 2015:n lukuisista uusista ominaisuuksista. Ennen ECMAScript 2015:n valmistumista kaksi suosituinta vaihtoehtoa olivat CommonJS-työryhmän moduulistandardi ja AMD-moduulistandardi. Karkeasti voidaan sanoa, että CommonJS-standardia käytetään pääsääntöisesti palvelinpuolella (Node.js), AMD:ta puolestaan selainpuolella esimerkiksi Require.js-kirjaston avulla. (2ality 2014)

Tässä työssä tulemme käyttämään ECMAScript 2015:n määrittelemiä moduuleja, emmekä perehdy tarkemmin muihin variaatioihin. Koska selaimet eivät ole vielä täysin yhteensopivia moduulien kanssa, joudumme kokoamaan ne yhteen kimpaleeseen (engl. chunk) Node.js:n päällä toimivan Webpack-työkalun avulla. Sekä Node.js että Webpack käsitellään yksityiskohtaisemmin seuraavassa kappaleessa.

2.3 Yksikkötestattavuus

Yksikkötesteistä puhuttaessa ”yksikkö” voi tarkoittaa useita eri asioita. Se voi olla esimerkiksi funktio, luokan metodi, kokonainen luokka tai mikä tahansa muu yhteensopiva kokonaisuus, jonka oikeanlainen toiminta voidaan vahvistaa testeillä (Art of Unit Testing 2011). Testi puolestaan on pala ohjelmakoodia, joka ajaa testattavan yksikön toiminnallisuuden läpi (esimerkiksi luo uuden olion luokasta tai kutsuu funktiota) ja varmistaa, että kaikki toimii halutulla tavalla.

Yksikkötestauksen kaksi tärkeintä tavoitetta on auttaa ohjelmistokehittäjiä suunnittelemaan järkeviä ja toimivia rakenteita kehityksen aikana sekä tarjota suojaa tahattomia regressioita vastaan (Simple Programmer 2010).

Opinnäytetyön lähtötilanteessa Wilman *addevent.js*-kirjaston koodi on lähes täysin yksikkötestaamatonta. Koska kirjasto sisältää paljon Wilman toiminnan kannalta kriittistä koodia, on ensisijaisen tärkeää suojata sen oikeellinen toiminta yksikkötesteillä. Tämä on yksi tärkeimpiä opinnäytetyössä käsiteltäviä uudistuskohteita.

Wilmassa JavaScriptin yksikkötestaamiseen käytetään Node.js:n päällä toimivia työkaluja; tärkeimmät näistä ovat Karma-testaustyökalu ja Jasmine-testauskehys. Molemmat työkalut esitellään Node.js:n lisäksi tarkemmin seuraavassa kappaleessa.

2.4 ECMAScript 2015

ECMAScript on Ecma Internationalin laatima ja ylläpitämä ohjelmointikielen määritys. Useiden eri sovellusten JavaScript-moottorit toteutetaan tämän määrityksen pohjalta. Esimerkiksi Mozilla SpiderMonkey ja Google V8 kuuluvat näihin. (InnoArchitech 2014)

ECMAScript 2015, joka tunnetaan myös epävirallisesti nimellä ECMAScript 6, on laaja päivitys ECMAScript-määritykseen. Se määrittelee valtavan kirjon uusia ominaisuuksia. Muutamia näistä ovat nuolifunktiot, *let*- ja *const*-muuttujat sekä moduulit. Lisäksi kyseessä on ensimmäinen versio, joka tukee perinteistä lähestymistapaa olio-ohjelmointiin.

Koodin yksinkertaistamiseksi, turvallisuuden parantamiseksi ja kehittäjien työn helpottamiseksi koodia tullaan työssä muokkaamaan siten, että se hyödyntää ECMAScript 2015:n ominaisuuksia. Edellä mainittujen hyötyjen lisäksi ECMAScript 2015 -uudistus vähentää koodin virhealttiutta mm. uusien muuttujatyyppien ansiosta.

2.4.1 Nuolifunktiot

Nuolifunktiot ovat uusi tapa määritellä funktioita JavaScriptissä. Kolme niiden tärkeintä hyötyä ovat yksinkertaisempi syntaksi, muutokset *this*-kontekstin käsittelyyn ja se, että ne ovat aina anonyymejä. (Felix Rieseberg 2015)

```
var ihmiset = [
    { nimi: "Jussi", ika: 21 },
    { nimi: "Liisa", ika: 19 },
    { nimi: "Matti", ika: 34 }
];

var nimetES5 = ihmiset.map(function (ihminen) {
    return ihminen.nimi;
});

let nimetES2015 = ihmiset.map(ihminen => ihminen.nimi);
```

Kuvio 3. Nuolifunktion kolmelta riviltä yhdelle riville tiivistämä toiminto

Kuvion 3 esimerkissä luodaan *Array*-tyyppinen muuttuja, *ihmiset*, joka sisältää kolme objektia, kukin ihmisen nimen ja iän. Muuttujiin *nimetES5* ja *nimetES2015* tallennetaan ihmisten nimet. Muuttujan *nimetES2015* muodostamiseen tarvitaan kuitenkin huomattavasti vähemmän koodia, kuin *nimetES5*:n muodostamiseen. Tämä on vain pieni esimerkki siitä, miten nuolifunktiosyntaksi helpottaa ja yksinkertaistaa koodin kirjoittamista.

2.4.2 Let ja const

JavaScriptissä on ollut sen alkuajoista asti käytössä *var*-tyypin muuttujat. *Var*-muuttujat saavat toiminta-alueekseen (engl. scope) sen funktion, jonka sisällä ne määritellään. Tämä tarkoittaa sitä, että esimerkiksi *if*-ehtolausekkeessa tai *for*-toistorakenteessa määritelty muuttuja on käytettävissä myös sen ulkopuolella.

Let-muuttujat korjaavat tämän ongelman pienentämällä muuttujien toiminta-aluetta. Ne eivät saa koko funktiota toiminta-alueekseen, vaan ainoastaan sen lohkon (engl. block), jossa ne määritellään.

```
if (true) {
    var luku = 4;
}

console.log(luku);
// --> 4
```

Kuvio 4. Var-muuttujan toiminta

Kuvion 4 esimerkissä määritellään uusi *var*-muuttuja ehtolausekkeen sisällä. Ehtolausekkeen jälkeen kutsutaan *console.log*-metodia, joka kirjoittaa konsoliin luvun 4. Muuttuja on siis näkyvissä myös ehtolausekkeen ulkopuolella. Seuraavassa kuviossa demonstroidaan *let*-muuttujan toimintaa vastaavassa tilanteessa.

```
if (true) {
    let luku = 4;
}

console.log(luku);
// --> ReferenceError: luku is not defined
```

Kuvio 5. Let-muuttujan toiminta

Kuvion 5 esimerkissä ainoa ero kuvion 4 koodiin on se, että *var*-avainsana on korvattu avainsanalla *let*. Kun yritämme kirjoittaa lukua ehtolausekkeen ulkopuolella konsoliin, saamme vastaukseksi vain virheen; muuttujaa *luku* ei ole määritetty. Sen toiminta-alue rajoittuu ehtolausekkeen sisään.

Const-muuttujat käyttäytyvät muiden ohjelmointikielien tapaisesti; kun ne määritellään ja niille annetaan arvo, niitä ei voi enää muokata. Muuttujan arvo on pysyvä.

```
const vekotin = "iPhone";

vekotin = "Android";
// --> invalid assignment to const `vekotin`
```

Kuvio 6. Const-muuttujan toiminta

Kuviossa 6 esitellään *const*-muuttujan toimintaa käytännössä. Muuttujaan *vekotin* asetetaan arvo ”iPhone”. Tämän jälkeen muuttujan *vekotin* arvoksi yritetään asettaa ”Android”, joka ei ole sallittua muuttujan tyypin ansiosta.

2.4.3 Luokat

JavaScript ei käytä perinteistä luokkiin perustuvaa perintämallia, vaan se on yksi harvoista ohjelmointikielistä, jotka käyttävät *prototyyppeihin* perustuvaa perintää.

”Usein tätä pidetään JavaScriptin eräänä suurimmista heikkouksista. Itse asiassa prototyyppipohjainen perintämalli on voimakkaampi kuin klassinen malli. Sen avulla voidaan mallintaa klassinen malli melko helposti. Toisin päin mallintaminen on huomattavasti vaikeampaa.” (GitHub 2017A)

ECMAScript 2015:n myötä myös JavaScriptissä voidaan käyttää muiden ohjelmointikielien kaltaista luokkamallia. Tämä on kuitenkin vain ns. ”syntaktista sokeria”, sillä todellisuudessa JavaScript-moottorit muuntavat luokkapohjaisen koodin ajonaikaisesti käyttämään prototyyppejä.

JavaScriptissä luokkia käytetään hyvin samankaltaisella tavalla, kuin muissa suosituissa ohjelmointikielissä. Uudet luokat määritellään avainsanalla *class* ja jotta uusia olioita voidaan luoda, ne tarvitsevat vähintään konstruktorin. Se määritellään avainsanalla *constructor***.** Luokat voivat myös periytyä muista luokista.

```
class PolkuPyora {

    constructor(merkki) {
        this.merkki = merkki;
        this.nopeus = 0;
        this.pystyssa = true;
    }

    polje() {
        this.nopeus = 10;
    }

    pysahdy() {
        this.nopeus = 0;
    }

    kaadu() {
        this.pysahdy();
        this.pystyssa = false;
    }

}

const fillari = new PolkuPyora("Helkama");
fillari.polje();
fillari.kaadu();
```

Kuvio 7. Luokan määrittely ja käyttäminen

Kuviossa 7 määritellään esimerkkiluokka, joka simuloi polkupyörää ja sen toiminnallisuutta. Sen metodeja ovat polkeminen, pysähtyminen ja kaatuminen. Luokan käyttöä demonstroidaan luomalla uusi Helkama-merkkinen polkupyörä, sijoittamalla se *fillari*-muuttujaan ja kutsumalla sen metodeja.

2.4.4 Moduulit

ECMAScript 2015:n moduulien tavoitteena oli löytää ratkaisu, johon sekä CommonJS- ja AMD-ratkaisujen käyttäjät olisivat tyytyväisiä. Syntaksin tulisi olla CommonJS:n tavoin yksinkertainen, mutta lisäksi sen tulisi AMD:n tavoin tukea asynkronista moduulien lataamista. (2ality 2014)

```
export const seponTervehdys = () => {
    console.log("Terve, olen Seppo Hevostinen.");
};
```

Kuvio 8. Funktion vienti

Kuvion 8 moduulissa määritellään uusi funktio, *seponTervehdys*, joka viedään muiden moduulien käytettäväksi *export*-avainsanalla. Yhdestä moduulista voi viedä funktioita, objekteja ja muuttujia eikä niiden määrää ole rajoitettu.

```
import * as testimoduuli from "./seppo.js";

testimoduuli.seponTervehdys();
// --> Terve, olen Seppo Hevostinen.
```

Kuvio 9. Funktion tuonti

Kuvion 9 esimerkissä tuodaan kuvio 8:ssa määritellyn moduulin sisältö avainsanalla *import*. Komento hakee kaiken *seppo.js*-tiedostosta viedyn sisällön ja asettaa sen nimeksi *testimoduuli*. Tämän jälkeen esimerkiksi *seponTervehdys*-funktiota voidaan kutsua funktiokutsulla *testimoduuli.seponTervehdys()*.

3 KÄYTETYT TEKNOLOGIAT

Tässä kappaleessa käsitellään Wilmassa käytettyjä teknologioita, joita tullaan käyttämään myös edellisessä kappaleessa käsiteltyjen uudistuskohteiden kehittämiseen. Kaiken perustana Wilman frontend-puolella on Node.js-alusta ja sen päällä ajettavat erilaiset työkalut. Kappaleessa käsitellään muun muassa tehtävien ajamiseen, yksikkötestien kirjoittamiseen, koodin kääntämiseen ja moduulien kokoamiseen liittyvien työkalujen periaatteita ja toimintaa.

3.1 Node.js

Node.js on alusta, jonka avulla voidaan kirjoittaa JavaScript-sovelluksia selainympäristön ulkopuolella. Perinteiseen selaimen sisällä tapahtuvaan JavaScript-kehitykseen erona on se, että DOM:ia tai muita selaimen ominaisuuksia ei ole käytettävissä. Ajon asynkronisuuden ja JavaScript-kielen ansiosta se on erittäin tehokas ja nopea alusta monenlaisille sovelluksille. (Herron 2011, 22-23)

Viime vuosien aikana Node.js:stä on tullut suosittu alusta palvelinohjelmistojen kehityksessä. Lisäksi Node.js on saavuttanut suuren suosion työpöytäsovellusten maailmassa; Node.js:ään perustuva Electron-sovelluskehys mahdollistaa perinteisten web-teknologioiden käyttämisen työpöytäsovellusten kirjoittamisessa. Muutamia esimerkkejä Electronia hyödyntävistä sovelluksista on Microsoft Visual Studio Code, GitHub Desktop ja Slack (Electron 2017).

Node.js:n oletusasennukseen kuuluu lisäksi npm-paketinhallintasovellus. Sen avulla voidaan rekisteröidä ja julkaista JavaScript-paketteja. Paketit tallentuvat npm-rekisteriin, josta käyttäjät voivat helposti asentaa ja käyttää niitä. (Juzer 2013, 15-16)

Opinnäytetyön puitteissa Node.js:ää tullaan käyttämään alustana, jonka päällä ajetaan erilaisia tehtäviä. Sitä käytetään muun muassa moduulien kokoamiseen sekä kehitys- että tuotantoympäristöihin, yksikkötestien ajoon ja kattavuusraporttien luontiin. Kaikki työssä käytetyt Node.js-työkalut asennetaan npm:n kautta.

3.2 Jasmine

Jasmine on JavaScript-koodin testaamiseen suunniteltu testauskehys, jonka avulla voidaan kirjoittaa kattavia testejä erilaisiin tarkoituksiin. Sitä voidaan ajaa sekä itsenäisesti että Pythonin, Rubyn ja Node.js:n päällä (Jasmine 2017A). Jasmine-testien rakennuspalikoita ovat *suite*, *spec* ja *expectation* (Jasmine 2017B).

Suite kasaa yksittäistä kokonaisuutta koskevat testit eräänlaiseen kansioon. Tämä kokonaisuus voi olla esimerkiksi luokka tai funktio. *Suite* määritellään funktiolla *describe(string, function)*. Ensimmäinen parametri on kuvaus *suiten* sisällöstä ja toinen on varsinainen sisältö funktion muodossa. (Jasmine 2017B)

Spec tarkoittaa yksittäistä testiä, joka varmistaa jonkin tietyn ominaisuuden oikeellisen toiminnan. Ne kirjoitetaan *suiten* toisen parametrin sisään käyttäen funktiokutsua *it(string, function)*. Ensimmäinen parametri on testin kuvaus ja toinen on varsinainen testi funktion muodossa. (Jasmine 2017B)

Expectation on asia, jonka tulee löytyä jokaisesta testistä, eli *specistä*. Niitä luodaan *expect()*-funktion avulla ja ne varmistavat, että testattava koodi käyttäytyy halutulla tavalla. Esimerkiksi *expect(true).toBe(true)* tarkoittaisi onnistunutta testiä.

```
describe("Method: trimRight", () => {

    it('should trim whitespaces from the right', () => {
        expect(stringHelpers.trimRight('Text    ')).toBe('Text');
    });

    it('should not trim whitespaces from the left', () => {
        expect(stringHelpers.trimRight('   Text    ')).toBe('   Text');
    });

});
```

Kuvio 10. Kokonainen Jasmine-suite kahdella testillä

Kuvion 10 esimerkissä määritellään täydellinen *suite*, jonka tarkoituksena on varmistaa *trimRight*-funktion oikeellinen toiminta. Funktion tarkoitus on poistaa

tyhjät välilyönnit merkkijonon lopusta, joka sille annetaan. Ensimmäinen testi varmistaa, että se poistaa välilyönnit merkkijonon lopusta. Toinen testi puolestaan varmistaa, että se ei poista välilyöntejä merkkijonon alusta.

3.3 Grunt

Grunt on Node.js-ympäristöön suunniteltu automaatiotyökalu. Se on tehtäväpohjainen työkalu, jonka avulla voidaan esimerkiksi minimoida alkuperäinen lähdekoodi jakelua varten tai ajaa yksikkötestejä (Pillora 2014, 21). Eri tehtäviä varten voidaan asentaa ylimääräisiä moduuleja esimerkiksi npm-pakettivaraston kautta.

Grunt konfiguroidaan suorittamaan haluttuja tehtäviä konfiguraatiotiedoston kautta, jota kutsutaan Gruntfileksi. Gruntfilessä mm. ladataan käytetyt moduulit sekä määritellään kyseisten moduulien asetukset ja eri tehtävät, joita Gruntin halutaan ajavan.

```
grunt.registerTask('build', ['babel', 'less', 'uglify:production', 'webpack:production']);
grunt.registerTask('bundle', ['webpack:production']);
grunt.registerTask('bundle-watch', ['webpack:development']);
grunt.registerTask('compress', ['uglify:withoutEs6']);
grunt.registerTask('wilma-watch-less', ['watch']);
grunt.registerTask('wilma-less', ['less']);
grunt.registerTask('wilma-jshint', ['jshint']);
grunt.registerTask('test', ['karma:module']);
grunt.registerTask('test-watch', ['karma:moduleWatch']);
grunt.registerTask('test-legacy', ['karma:independent']);
grunt.registerTask('test-legacy-watch', ['karma:independentWatch']);
grunt.registerTask('test-all', ['karma:module', 'karma:independent'])
grunt.registerTask('test-coverage', ['karma:moduleCoverage']);
grunt.registerTask('test-coverage-watch', ['karma:moduleCoverageWatch']);
```

Kuvio 11. Wilman Gruntfilessä määritetyt tehtävät

Wilman Gruntfilessä on määritetty yhteensä neljätoista erilaista tehtävää. Nämä tehtävät on listattu kuviossa 11. Erilaisia tehtäviä löytyy mm. yksikkötestien ajamiseen, koodin pakkaamiseen ja kattavuusraporttien luomiseen.

Taulukko 1. Grunt-tehtävät

wilma-less	Kääntää LESS-tiedostot CSS:ksi
compress	Pakkaa JavaScript-koodin minimoituihin tiedostoihin
bundle	Rakentaa moduuleista bundle-tiedostot Webpackin avulla
test-coverage	Ajaa yksikkötestit ja luo HTML-muotoisen kattavuusraportin

Taulukossa 1 esitellään muutamia oleellisia Gruntfilessä määritettyjä tehtäviä. Lisäksi muun muassa yksikkötesteihin liittyviin tehtäviin on saatavilla *watch*-tila. Yksikkötesteihin liittyvien tehtävien kohdalla tämä tarkoittaa sitä, että testien ajosta vastuussa oleva Karma tarkastelee jatkuvasti kooditiedostoja muutosten varalta ja suorittaa testit uudelleen muutosten tapahtuessa.

3.4 Karma

Karma on Node.js:n päällä toimiva yksikkötestaustyökalu. Karma toimii käynnistämällä tietokoneelle paikallisen web-palvelimen, jota vasten se ajaa halutut yksikkötestit. Testien tulokset tulostetaan konsoli-ikkunaan, josta kehittäjä näkee onnistuneesti suoritetut ja epäonnistuneet testit. (GitHub 2017B)

Lisäksi Karma mahdollistaa:

- testien ajamisen useita eri selaimia vasten
- useiden testauskehyksien käyttämisen
    - Yksi tuetuista kehyksistä on tässä työssä käytetty Jasmine
- kattavuusraporttien luomisen

```
START:
10 10 2017 19:08:49.803:INFO [karma]: Karma v1.4.1 server started at http:/
10 10 2017 19:08:49.806:INFO [launcher]: Launching browser PhantomJS with u
10 10 2017 19:08:49.832:INFO [launcher]: Starting browser PhantomJS
10 10 2017 19:08:53.161:INFO [PhantomJS 2.1.1 (Windows 8 0.0.0)]: Connecte
  Module: date
    Method: isLeapYear
      √ should return true if the year is a leap year
      √ should return false if the year is not a leap year

Finished in 0.01 secs / 0.005 secs @ 19:08:53 GMT+0300 (Suomen kesäaika)

SUMMARY:
√ 2 tests completed

Done.
```

Kuvio 12. Karman ajaminen konsoli-ikkunasta

Kuvion 12 esimerkissä suoritetaan yksikkötestien ajo Karmalla Gruntin avulla. Kuvion tekstit *Module: date* ja *Method: isLeapYear* tulevat Jasmine-kehyksen *suitelta*. Näiden otsikoiden alapuolelle tulostetaan Jasmine-kehyksen *specien*, eli yksittäisten testien tulokset, joko vihreällä tai punaisella. Vihreä teksti indikoi testin onnistumista ja punainen epäonnistumista.

/

91.17% Statements 1580/1733 **88.14%** Branches 840/953 **95.86%** Functions 324/338 **88.88%** Lines 1071/1205

50 statements, 21 functions, 96 branches Ignored

File		Statements		Branches		Functions		Lines	
commons/		96.15%	25/26	100%	10/10	100%	7/7	75%	3/4
commons/appexports/		85%	17/20	100%	8/8	50%	1/2	62.5%	5/8
commons/data-validators/		100%	30/30	100%	30/30	100%	2/2	100%	29/29
commons/environment/		100%	54/54	100%	25/25	100%	17/17	100%	27/27
commons/helpers/common/		88.46%	46/52	83.78%	31/37	100%	6/6	88.24%	45/51
commons/helpers/date/		100%	86/86	96.55%	56/58	100%	12/12	100%	85/85
commons/helpers/form/		100%	41/41	100%	21/21	90%	9/10	100%	40/40
commons/helpers/modal/		100%	23/23	100%	10/10	100%	1/1	100%	22/22
commons/helpers/sorter/		87.1%	27/31	90.91%	20/22	100%	3/3	100%	19/19
commons/helpers/string/		100%	17/17	100%	13/13	100%	3/3	100%	16/16
commons/history-state/		100%	37/37	88.89%	16/18	100%	10/10	100%	14/14

Kuvio 13. Yksikkötestien kattavuusraportti

Karman ajon yhteydessä luotua kattavuusraporttia esitellään kuviossa 13. Raportin yläreunassa kerrotaan kaikkien ajettujen testien tilastot; kuvion testiajossa on katettu yhteensä 91,17 % kaikista lauseista (engl. statement), 88,14 % haaroista (engl. branch) ja niin edelleen. Tarkemmat, moduulikohtaiset tiedot löytyvät eriteltynä taulukosta.

3.5 Babel

Koska selainten ja muiden JavaScriptiä tukevien ympäristöjen ECMAScript-tuki laahaa jäljessä standardin ominaisuuksista, ohjelmoijat eivät voi käyttää kielen viimeisimpiä ominaisuuksia heti niiden tullessa saataville. Esimerkiksi Internet Explorer 11 -selain tukee vain 11 % kaikista ECMAScript 2015:n ominaisuuksista, kun taas Google Chrome 62 -selaimen tuki on 97 %. (GitHub 2017C).

Babelin avulla voidaan muuntaa (engl. transpile) ECMAScript 2015 -koodi (tai uudempi) takaisin JavaScript-moottorien ymmärtämäksi ECMAScript 5:ksi (Kleo Petrov 2016). Tämä tarkoittaa sitä, että ohjelmoija voi käyttää kielen viimeisimpiä ominaisuuksia tuotteen kehitysvaiheessa välittämättä esimerkiksi selaintuen kattavuudesta.

```
class Auto {
    constructor(nopeus) {
      this.nopeus = nopeus;
    }
}
```

```
function _classCallCheck(instance,
Constructor) { if (!(instance
instanceof Constructor)) { throw
new TypeError("Cannot call a class
as a function"); } }

var Auto = function Auto(nopeus) {
  _classCallCheck(this, Auto);

  this.nopeus = nopeus;
};
```

Kuvio 14. Babel-käännös ECMAScript 2015:stä ECMAScript 5:ksi

Babelin toimintaa esitellään luokkien avulla kuviossa 14. Vasemmalla puolella määritellään uusi luokka, *Auto*, käyttäen ECMAScript 2015:n uutta luokkasyntaksia. Kuvion oikealla puolella on Babelin tuottama selainyhteensopiva koodi, joka käyttää modernin luokkasyntaksin sijaan perinteistä prototyyppisyntaksia.

3.6 Webpack

Opinnäytetyön kirjoitushetkellä selaimet eivät osaa vielä täysin tulkata JavaScript-moduuleja. Tästä syystä niitä käyttävissä web-sovelluksissa täytyy ajaa ennen tuotantovaihetta jonkinlainen kokoamisprosessi, joka luo sovelluksesta lopullisen, selaimissa ajettavan jakeluversion. Yhteensopivuuden lisäksi sovelluksen kokoamisesta on se hyöty, että asiakasselaimet välttyvät pahimmillaan useiden satojen http-pyyntöjen tekemiseltä (Microsoft 2012). Opinnäytetyön kirjoitushetkellä ylivoimaisesti suosituin kokoamistyökalu JavaScript-maailmassa oli Webpack.

Kokoamisprosessi aloitetaan Webpackille määritetystä aloituspisteestä, joita voi olla useita. Webpack luo yhden koodia sisältävän kimpaleen (engl. chunk) per aloituspiste. Kimpaleiden riippuvuuksista luodaan ajonaikaisesti riippuvuuskaavio, jotta Webpack tietää mitä kaikkea niihin täytyy sisällyttää (Webpack 2017A).

Aloituspisteistä rakennettavien kimpaleiden lisäksi luodaan yksi ylimääräinen kimpale. Tämä kimpale rakennetaan ylimääräisen Webpack-lisäosan avulla ja se sisältää sellaista koodia, joka jaetaan muiden kimpaleiden kesken. Tästä on se hyöty, että asiakasselaimet voivat tallentaa jaetun kimpaleen välimuistiinsa, eikä sitä täydy ladata jokaisen kimpaleen kohdalla uudestaan (Webpack 2017B). Jaettu kimpale sisältää mm. tässä työssä toteutettavat moduulit.

```
$ grunt bundle-watch
Running "webpack:developVersion: webpack 2.2.1
                     Asset     Size  Chunks                    Chunk Names
       ./messages.bundle.js   217 kB       0  [emitted]         messages
./family-settings.bundle.js  7.08 kB       1  [emitted]         family-settings
        ./specedu.bundle.js  31.7 kB       2  [emitted]         specedu
           ./fake.bundle.js  1.03 kB       3  [emitted]         fake
          commons.bundle.js   951 kB       4  [emitted]  [big]  commons
```

Kuvio 15. Webpackin ajo kehitystilassa Gruntin avulla

Gruntin avulla ajettua Webpack-kokoamista on esitelty kuviossa 15. Webpack muodostaa jokaisesta aloituspisteestä riippuvuuskaaviot ja suorittaa kokoamisen. Tuloksena tästä on useita kimpaleita, joista jokainen edustaa tiettyä näkymää Wilmassa.

4 TOTEUTUS

Tässä kappaleessa käsitellään työn teknistä toteutusta vaihe vaiheelta. Toteutus-kappale on jaettu kolmeen vaiheeseen, jotka ovat suunnittelu-, ohjelmointi- ja yhteensopivuusvaihe.

Suunnitteluvaiheessa käsitellään toteutettavien moduulien rakennetta ja luodaan pohja niiden toiminnalle. Ohjelmointivaiheessa siirretään vanhaa koodia moduuleihin, kirjoitetaan yksikkötestejä ja refaktoroidaan koodia tarpeen vaatiessa. Yhteensopivuusvaiheessa varmistetaan, että uusi ja vanha koodi toimivat oikein toistensa kanssa.

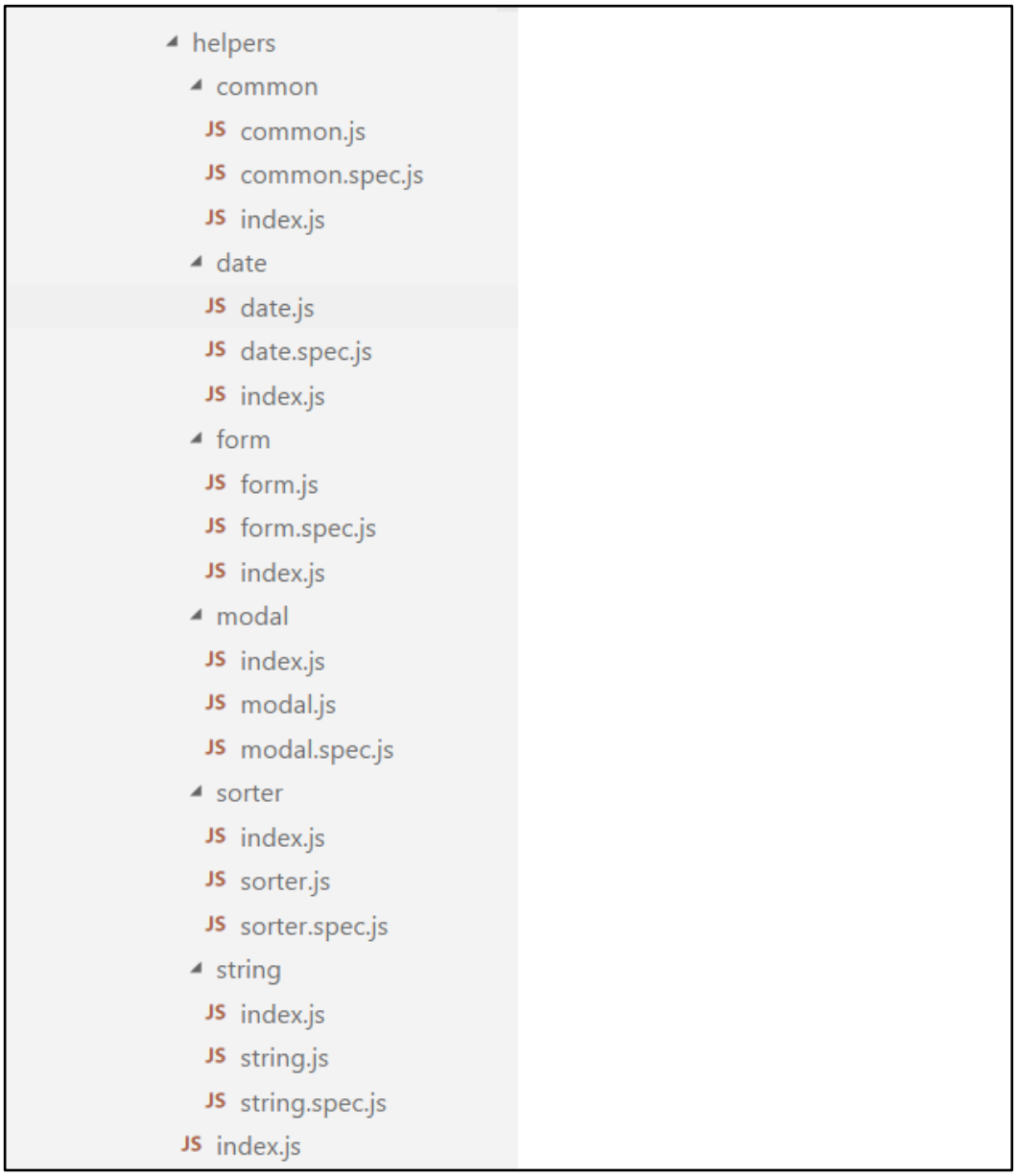
4.1 Suunnittelu

Suuri osa uudistettavan kirjaston, *addevent.js*:n, koodista koostuu yksinkertaisista apufunktioista, jotka ottavat vastaan tiedon X ja palauttavat tiedon Y. Työssä keskitytään uudistamaan näitä apufunktioita, koska pääsääntöisesti niiden kaltainen koodi on helposti testattavaa. Helppo testattavuus johtuu siitä, että koodilla ei ole paljoa ulkopuolisia riippuvuuksia ja sen tehtävät ovat helposti määriteltävissä.

Taulukko 2. Moduulien tehtävänjako

Nimi	**Tarkoitus**
helpers	Päämoduuli, joka kokoaa muut moduulit yhteen
date	Aikaan liittyvän datan käsittely
form	HTML-lomakkeiden käsittely
modal	Ponnahdusikkunoiden luonti ja käsittely
sorter	Datan järjestäminen
string	Merkkijonojen käsittely
common	Sisältö, joka ei sovi mihinkään muuhun moduuliin

Taulukossa 2 esitellään työssä rakennettavat moduulit ja niiden tarkoitus. Itsenäisiä moduuleja tullaan toteuttamaan kuusi erilaista, joista viisi on rajattu tiettyyn käyttötarkoitukseen. Kuudes, ylimääräinen moduuli tarkoitettu sisällölle, joka ei suoraan sovi muihin moduuleihin. Koko rakenteen huipulla on päätason *helpers*-moduuli, joka kokoaa kaikki alamoduulit yhdeksi kokonaisuudeksi.



Kuvio 16. Moduulikokonaisuuden kansiorakenne

Kuviosta 16 käy ilmi valmis moduulien kansiorakenne. Jokainen moduuli sijaitsee omassa alikansiossaan *helpers*-kansion alla. Kukin alikansioista sisältää kolme tiedostoa; itse koodin (*.js*-pääte), yksikkötestit (*.spec.js*-pääte) ja *index.js*-

tiedoston. Myös *helpers*-kansio sisältää moduulikansioiden lisäksi oman *index.js*-tiedostonsa. Näiden tiedostojen sisältöä ja tarkoitusta käsitellään seuraavaksi.

```
export const string = {
    trimRight(str) {
        return str.replace(/\s+$/g, "");
    }
};
```

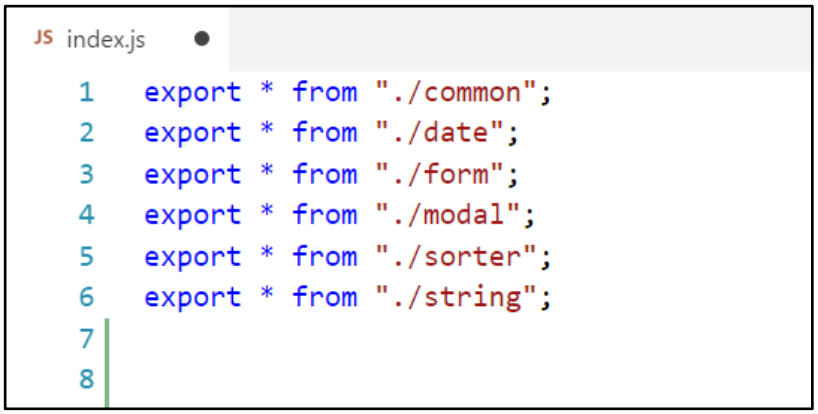
Kuvio 17. Yksittäisen moduulin rakenne

Kuviossa 17 esitellään *string.js*-tiedostoon uusi objekti, *string*, joka käyttäytyy minkä tahansa muun JavaScript-objektin tavoin. Tämä objekti julkaistaan (avainsana *export)* yksittäisenä moduulina muun koodin käytettäväksi. Lisäksi objekti määritellään muuttumattomaksi *const*-avainsanalla. Sen sisään voidaan kirjoittaa funktioita pilkuin eroteltuna. Kuvion objekti sisältää esimerkin vuoksi yhden funktion, *trimRight*.

```
JS index.js ●
1   export * from "./string";
2
3
4
```

Kuvio 18. string-moduulin *index.js*

Jokaiseen moduuliin kuuluvan *index.js*-tiedoston tehtävä on sangen yksinkertainen. Se kokoaa yhdestä kansiosta yhteen kaiken sisällön, joka halutaan julkaista muun koodin käytettäväksi. Esimerkki tästä löytyy kuviosta 18. *String*-moduulin tapauksessa halutaan julkaista vain *string.js*-tiedoston sisältö, kun taas *string.spec.js*-tiedostoa ei julkaista. Sitä tarvitaan vain yksikkötestien ajossa ja kattavuusraporttien luonnissa.



Kuvio 19. *Helpers*-kansion *index.js*

Helpers-kansion *index.js* kokoaa yhteen kuuden alamoduulin *index.js*-tiedostot ja julkaisee ne sovelluksenlaajuisesti kuvion 19 mukaisesti. Tämän *index.js*:n ei tarvitse määritellä haettavan sisällön täydellisiä tiedostonimiä, vaan pelkkä kansion nimi riittää haettavan kansion *index.js*:n ansiosta.

Kun rakenne toimii oikein, muualta koodista voidaan hakea vaihtoehtoisesti joko *helpers*-moduuli kokonaisuudessaan tai jokin sen alamoduuleista käytettäväksi yksinkertaisella *import*-komennolla.

```
import * as helpers from "../helpers";

helpers.string.trimRight("  merkkimössö    ");
// --> "  merkkimössö"
```

Kuvio 20. Helpers-moduulin käyttö

Kuvion 20 esimerkissä demonstroidaan toimivan *helpers*-moduulin käyttöä. Käyttöön voidaan tuoda joko *helpers*-moduuli kokonaisuudessaan, kuten kuviossa, tai vain jokin sen alamoduuleista.

Seuraava vaihe työn toteutuksessa on tässä luvussa toteutettujen moduulien kansoittaminen funktioilla.

4.2 Ohjelmointi

Tässä luvussa käsitellään toteutuksen ohjelmointivaihetta. Uudistamisprosessi etenee vaiheittain. Ensin alkuperäinen koodi siirretään uuteen kotiinsa moduuli-puolelle sellaisenaan. Tämän jälkeen sen tehtävät määritellään ja sille kirjoitetaan yksikkötestit. Koodin ECMAScript 2015 -refaktorointi tehdään vasta lopuksi, jotta koodin oikeellisesta toiminnasta voidaan olla koko ajan varmoja toimivien yksikkötestien ansiosta. Luvussa käydään läpi tämä prosessi kolmen yksinkertaisen funktion kohdalla.

4.2.1 getLokalisointiFromElement

getLokalisointiFromElement-funktion tehtävä on hakea DOM:sta oikealla kielellä lokalisoitu teksti lokalisoinnin tunnistenumeron perusteella. Funktiolle voidaan myös antaa pino HTML-elementtejä (*stack*), josta lokalisointia etsitään koko DOM:n sijaan. Funktio tullaan sijoittamaan osaksi *common*-moduulia.

```
function getLokalisointiFromElement(id, stack) {
    var elm;
    if (typeof stack === 'undefined') stack = null;

    if (stack === null) elm = $('#lok-' + id);
    else elm = stack.find('#lok-' + id);

    if (elm.length) return elm.text();
    else return '';
}
```

Kuvio 21. getLokalisointiFromElement, alkuperäinen koodi

Kuviossa 21 on uudistettava funktio alkuperäisessä tilassaan *addevent.js*:ssä. Sen suurin puute on vaikealukuisuus; *if*-lauseet on kirjoitettu kokonaan yhdelle riville ja ECMAScript 5:n puutteiden ansiosta pinon olemassaoloa täytyy erikseen tarkistella ylimääräisillä *if*-lauseilla. Ennen refaktorointia funktiolle kirjoitetaan yksikkötestit.

getLokalisointiFromElement:lle voidaan kirjoittaa ainakin kolme testiä:

- Lokalisointi on olemassa
    a. Testi 1: Lokalisoinnin tulisi löytyä annetusta pinosta
    b. Testi 2: Lokalisoinnin tulisi löytyä, vaikka pinoa ei anneta
- Lokalisointia ei ole olemassa
    a. Testi 3: Lokalisointia ei tulisi löytyä millään parametreilla

```
describe("Method: getLokalisointiFromElement", () => {
    const LOK_1 = "Tämä on turhaa tekstiä.";
    const LOK_2 = "Tämä on erittäin tärkeää tekstiä."
    const LOK_3 = "Tämä on vielä tärkeämpää tekstiä."

    beforeEach(() => {
        window.document.body.innerHTML = "";
    });

    const generateDummyElement = (locNum, text) => {
        return $("<span />")
            .attr("id", `lok-${locNum}`)
            .text(text);
    };

});
```

Kuvio 22. getLokalisointiFromElement, testien pohja

Kuvion 22 koodilohkossa on määritelty yllä lueteltuja testejä varten tarvittava pohjarakenne. Kuvion *const*-muuttujat sisältävät kolme merkkijonoa, jotka asetetaan testeissä virtuaalisen DOM:n sisään etsimistä varten. Jasminen *beforeEach*-lohko suorittaa DOM:n tyhjentämisen ennen jokaista testiä. *generateDummyElement* on funktio, jota käytetään testeissä DOM:iin asetettavien elementtien luonnissa.

```
it("should find the correct text when a stack is used", () => {
    const puppu = generateDummyElement(2, LOK_2)
    const haettava = generateDummyElement(3, LOK_3);
    const stack = $("<div />")
        .append(puppu)
        .append(haettava)
        .appendTo(window.document.body);
    expect(commonHelpers.getLokalisointiFromElement(3, stack)).toEqual(LOK_3);
});
it("should find the correct text when a stack is not used", () => {
    const puppu = generateDummyElement(3, LOK_3).appendTo(window.document.body);
    const haettava = generateDummyElement(2, LOK_2).appendTo(window.document.body)
    expect(commonHelpers.getLokalisointiFromElement(2)).toEqual(LOK_2);
});
it("should return an empty string if the loc was not found", () => {
    const puppu1 = generateDummyElement(1, LOK_1);
    const puppu2 = generateDummyElement(2, LOK_2);
    const stack = $("<div />")
        .append(puppu1)
        .append(puppu2);
    expect(commonHelpers.getLokalisointiFromElement(3, stack)).toEqual("");
});
```

Kuvio 23. getLokalisointiFromElement, valmiit yksikkötestit

Kuviossa 23 on kolme valmista testiä, kukin oman Jasminen *it*-lohkonsa sisällä. Testit on kirjoitettu ylhäältä alas lukien samaan järjestykseen, kuin aiemmin esitellyt kolme testiä. Ensimmäinen testi varmistaa, että elementti löytyy annetusta pinosta ja toinen testi varmistaa, että elementti löytyy, vaikka pinoa ei anneta. Kolmas testi puolestaan varmistaa, että elementtiä ei löydy, jos se ei ole osa DOM:ia.

```
Method: getLokalisointiFromElement
  ✓ should find the correct text when a stack is used
  ✓ should find the correct text when a stack is not used
  ✓ should return an empty string if the loc was not found
```

Kuvio 24. getLokalisointiFromElement, onnistunut testien ajo

Kuviossa 24 näkyy konsoli-ikkunassa Karman tulostama loki onnistuneesta testien ajosta. Tekstin vihreä väri kertoo siitä, että kaikki kolme testiä toimivat oikein.

```
getLokalisointiFromElement(id, stack = null) {
    let elm;

    if (stack === null) {
        elm = $("#lok-" + id);
    } else {
        elm = stack.find("#lok-" + id);
    }

    return elm.length ? elm.text() : "";
},
```

Kuvio 25. getLokalisointiFromElement, lopullinen koodi

Alkuperäinen koodi on kuviossa 25 uudistettu käyttämään ECMAScript 2015:n ominaisuuksia. Pinon, eli *stackin*, oletusarvoksi voidaan määrittää funktion parametrilistassa *null*, jos ei sitä anneta funktiolle. Tämän avulla vältytään yhdeltä *if*-tarkistukselta. Lisäksi *elm* on *let*-muuttuja, *if*-lauseet on rivitetty luettavammiksi ja viimeinen *return*-lause on yksinkertaisuutensa vuoksi yhdellä rivillä.

4.2.2 jsDateToDelphiDateTime

Delphi-ohjelmointiympäristössä, jolla Wilman palvelin on kirjoitettu, aika-arvot käsitellään liukulukuina. Delphin ajanlasku alkaa 31.12.1899 klo 00:00 ja jokainen kulunut päivä kasvattaa aika-arvoa yhdellä (Delphi Basics 2017). *jsDateToDelphiDateTime*-funktion tarkoitus on muuttaa JavaScriptin *Date*-olion arvo Delphin ymmärtämään muotoon.

```
function jsDateToDelphiDateTime(jsdate) {
    jsdate = jsdate.getTime() - (jsdate.getTimezoneOffset() * 60000);
    var oneDayMs = 24 * 60 * 60 * 1000;
    var unixDate = strToDateTime('01.01.1970 00:00');
    var delphiDate = strToDateTime('31.12.1899 00:00');
    var diffDays = Math.round(
        Math.abs(
            (delphiDate.getTime() - unixDate.getTime()) / (oneDayMs)
        )
    );
    return ((jsdate / oneDayMs) + diffDays + 1);
}
```

Kuvio 26. jsDateToDelphiDateTime, alkuperäinen koodi

Kuviossa 26 esitellään alkuperäinen funktion koodi, joka siirretään moduulipuolelle. Koodissa ei sellaisenaan ole mitään suurempia vikoja, joten se ei tule kokemaan suurta muutosta ECMAScript 2015:n myötä.

```
describe("Method: jsDateToDelphiDateTime", () => {
    it("should return a valid Delphi timestamp", () => {
        // Tue, 23 May 1978 00:00:00 +0000
        const STAMP_1_UTC = 264643200000;
        const STAMP_1_DELPHI = 28632.125;

        const STAMP_2_RFC = "Sat, 05 Jun 2004 19:08:07 +0300";
        const STAMP_2_DELPHI = 38143.79730324074;

        expect(dateHelpers.jsDateToDelphiDateTime(new Date(STAMP_1_UTC)))
            .toEqual(STAMP_1_DELPHI);
        expect(dateHelpers.jsDateToDelphiDateTime(new Date(STAMP_2_RFC)))
            .toEqual(STAMP_2_DELPHI);
    });
});
```

Kuvio 27. jsDateToDelphiDateTime, valmis yksikkötesti

jsDateToDelphiDateTime-funktiolla on vain yksi, melko yksinkertainen tehtävä, joten sen riittävän kattavaan testaamiseen riittää yksi testi. Testi esitellään kuviossa 27. Ensin testissä luodaan kaksi aikaleimaparia, molemmissa pareissa yksi JavaScriptille ja yksi Delphille. Seuraavaksi testattavaa funktiota kutsutaan aikaleimaan täsmäävällä *Date*-objektilla ja varmistetaan, että paluuarvo täsmää vastaavan Delphi-aikaleiman kanssa.

```
jsDateToDelphiDateTime(jsDate) {
    const
        ONE_DAY_MS = 24 * 60 * 60 * 1000,
        UNIX = this.strToDateTime("01.01.1970 00:00"),
        DELPHI = this.strToDateTime("31.12.1899 00:00"),
        DIFF_DAYS = Math.round(
            Math.abs(
                (DELPHI.getTime() - UNIX.getTime()) / (ONE_DAY_MS)
            )
        );

    jsDate = jsDate.getTime() - (jsDate.getTimezoneOffset() * 60000);
    return ((jsDate / ONE_DAY_MS) + DIFF_DAYS + 1);
},
```

Kuvio 28. jsDateToDelphiDateTime, lopullinen koodi

Kuviossa 28 esitellään funktion lopullinen versio. *jsDateToDelphiDateTime*-funktio ei uudistamisen jälkeen käytä juurikaan uusia ECMAScript-ominaisuuksia. Ainoa ero on *var*-muuttujista *const*-muuttujiin siirtyminen.

4.2.3 datetimeIsInPast

datetimeIsInPast-funktio ottaa parametrina *Date*-objektin, tarkistaa sijaitseeko sen arvo menneisyydessä ja palauttaa joko *true* tai *false* tuloksesta riippuen.

```
function datetimeIsInPast(dt) {
    var now = new Date().getTime();
    var dtMs = dt.getTime();
    if (now > dtMs) {
        return true;
    } else {
        return false;
    }
}
```

Kuvio 29. datetimeIsInPast, alkuperäinen koodi

Kuvion 29 funktio ottaa parametrina *Date*-objektin. Tämän jälkeen luodaan kaksi muuttujaa, joista toiseen haetaan nykyhetken UTC-aikaleima ja toiseen paramet-

rina annetun *Date*-objektin UTC-aikaleima. Funktio palauttaa vastauksen näiden aikaleimojen vertailun avulla.

```
describe("Method: datetimeIsInPast", () => {

    it("should return false if the date is in the future", () => {
        const STAMP = new Date().getTime() + (60 * 60 * 1000);
        const date = new Date(STAMP);
        expect(dateHelpers.datetimeIsInPast(date)).toBe(false);
    });

    it("should return true if the date is in the past", () => {
        const date = new Date("2014-06-06T12:40:40");
        expect(dateHelpers.datetimeIsInPast(date)).toBe(true);
    });

});
```

Kuvio 30. datetimeIsInPast, valmiit yksikkötestit

Kuviossa 30 määritellään *datetimeIsInPast*-funktiolle kaksi testiä; testi, joka tarkistaa *false*-palautusarvon ja testi, joka varmistaa *true*-palautusarvon.

Ensimmäisessä testissä varmistetaan *false*-paluuarvo. Ensin luodaan *STAMP*, jonka avulla luodaan uusi *Date*-objekti, joka sijoittuu arvoltaan tunnin tulevaisuuteen. Tämän jälkeen tarkistetaan, että testattava funktio palauttaa *false*, koska sille annettu arvo on tulevaisuudessa. Toinen testi on toiminnaltaan samanlainen, mutta funktiolle annettava *Date*-objekti osoittaa vuoden 2014 kesäkuuhun, jolloin tuloksen pitää olla *true*.

```
datetimeIsInPast(date) {
    return new Date().getTime() > date.getTime();
},
```

Kuvio 31. datetimeIsInPast, lopullinen koodi

Kuten kuviosta 31 ja siinä olevan koodin lyhyestä muodosta käy ilmi, funktion alkuperäinen muoto oli epätehokas ja liian pitkä. Siinä luotiin turhaan useita *Date*-objekteja, tallennettiin millisekunteja muuttujiin ja vertailtiin niiden arvoja. Kaik-

ki tämä voidaan tehdä yhdellä rivillä, joka vertaa kahden *Date*-objektin arvoa ja palauttaa tuloksen.

4.3 Yhteensopivuus

Tämä luku käsittelee työn toteutusvaiheen viimeistä osaa, eli vaihetta, jossa varmistetaan uuden ja vanhan koodin yhteensopivuus ja toimivuus. Tämä vaatii muutoksia mm. Wilman palvelimeen, Webpackin kimpaleisiin ja HTML-mallinetiedostoihin.

4.3.1 JavaScriptEndpoint

Työn lähtötilanteessa vain tietyt näkymät Wilmassa pääsevät käyttämään moduulipuolen komponentteja. Tämä johtuu siitä, että Webpackin kimpaleet sisällytetään HTML-dokumentin *head*-osioon vain sellaisilla sivuilla, joiden HTML-malline varta vasten määrittelee käytettävän kimpaleen.

Wilman palvelin käyttää *TWebServiceHandler*-luokkaa käyttäjän istunnon tietojen tallentamiseen. Muun muassa käyttäjälle luotavassa sivussa käytettävä HTML-malline on yksi näistä tiedoista. *TWebServiceHandler*-luokkaan on tarpeen toteuttaa uusi metodi, jota käytetään HTML-mallinetiedoston asettamisen lisäksi halutun Webpack-kimpaleen määrittämiseksi.

```
procedure TWebServiceHandler.SetJavaScriptEndpoint(const Value: string);
begin
  FJavaScriptEndpoint := Value;
  FVariables['JavaScriptEndpoint'] := Value;
end;
```

Kuvio 32. SetJavaScriptEndpoint-metodi

Kuviossa 32 esitellyn muutoksen myötä *TWebServiceHandler*-luokalle voidaan määrittää mallinetiedoston lisäksi uusi ominaisuus, *FJavaScriptEndpoint.* Ominaisuutta tulisi käyttää vain, jos käyttäjä pyytää tiettyä moduulinäkymää (engl. view) Wilmasta. Muussa tapauksessa HTML-dokumenttiin upotetaan Webpackin *fake*-kimpale, joka sisältää mm. työssä toteutetut *helpers*-moduulit.

4.3.2 Hakijafunktiot

Toimiessaan oikein *helpers*-moduuli löytyy selaimen *Window*-objektin sisältä. Jos ohjelmiston kehittäjä haluaa käyttää esimerkiksi merkkijonoihin liittyvää *helpers*-funktiota, ne löytyisi sijainnista *Window.app.exports.helpers.string*. Tämän monitasoisen hierarkian vuoksi on hyvä olla olemassa jonkinlainen käyttöä helpottava funktio.

```
function helpers() {
    if (window.app && window.app.exports && window.app.exports.helpers) {
        return window.app.exports.helpers;
    }
    throw "Helpers module not found.";
}
```

Kuvio 33. Helpers-hakija

Kuten kuviosta 33 näkyy, kyseessä on melko yksinkertainen funktio, jonka tarkoituksena on vain lyhentää moduulien käyttöön tarvittavan tekstin määrää. Se tarkistaa, että kaikki tarvittavat tasot ovat olemassa ja palauttaa lopulta *helpers*-objektin. Jos kaikkia tarvittavia hierarkian paloja ei löydy, nostetaan asiasta poikkeus.

4.3.3 Viittausten korjaus

Työssä uudistetusta kirjastosta on työn ohella siirretty runsaasti koodia moduulipuolelle. Tästä syystä kaikki vanha koodi muualla Wilmassa, joka viittasi näihin funktioihin, ei enää toimi. Niiden viittaukset täytyy käydä systemaattisesti läpi ja korjata viittaamaan vastaavaan moduulipuolen funktioon.

```
if (!datetimeIsInPast(date) && !cancelButtonVisible) {
```

Kuvio 34. Vanhentunut funktioviittaus

Kuviossa 34 viitataan edellisessä luvussa käsiteltyyn *datetimeIsInPast*-funktioon. Tämä koodi aiheuttaisi ajonaikaisesti virheen, koska *datetimeIsInPast* on arvoltaan *undefined*, eli sitä ei ole olemassa.

```
if (!helpers().date.datetimeIsInPast(date) && !cancelButtonVisible) {
```

Kuvio 35. Päivitetty funktioviittaus

Kuvion 34 koodin on korjattu kuviossa 35 uusi funktioviittaus. Se käyttää uutta, *addevent.js*:n alkuun lisättyä hakijafunktiota ja tarkemmin sen alta *date*-moduulia. Koodi toimii tämän muutoksen jälkeen kuten ennenkin.

5 YHTEENVETO

Tämän opinnäytetyön aiheena oli uudistaa vaasalaisen Visma InCommunity Oy:n Wilma-tuotteen käyttämää JavaScript-koodia. Koodikirjaston sisältö ja rakenne olivat päässeet auttamatta vanhanaikaisiksi, ja se kaipasi uudistusta monella osa-alueella. Työn toisessa luvussa käytiin läpi ohjelmakoodia, sen puutteita ja kehityskohteita. Työssä toteutettaviksi kehityskohteiksi valittiin ECMAScript 5:stä ECMAScript 2015 -standardiin siirtyminen, koodin jakaminen loogisiin moduuleihin ja yksikkötestien kattavuuden kasvattaminen.

Työn tavoitteena ei ollut vain teknisen tuotoksen toteuttaminen. Lisäksi tavoitteisiin kuuluivat sekä oman että kollegoiden asiantuntemuksen kasvattaminen moderneista web-teknologioista, Node.js-alustasta ja testattavan koodin kirjoittamisesta.

Opinnäytetyön aikana tuli selväksi, ettei ole kovin helppoa sovittaa vanhoja ja uusia ohjelmointitekniikoita yhteen. Vaikka varsinainen tekninen toteutus sujuikin melko hyvin ilman suurempia haasteita, aloin kyseenalaistamaan projektin hyötyjä ja sitä, kannattaako työssä uudistettua koodia ylipäätään uudistaa.

Suuri osa työssä uudistetuista funktioista on hyvin spesifejä, tiettyyn tarkoitukseen kirjoitettuja funktioita, joita on käytetty koko ohjelman lähdekoodissa vain muutamia kertoja. Mieltäni askarrutti muun muassa se, onko tällaisen koodin siirtämisestä enemmän hyötyä kuin haittaa. Funktioita todennäköisesti ei tulla käyttämään uudestaan uusissa ohjelmaan kirjoitettavissa ominaisuuksissa, mutta ne elävät ikuisesti moduuleissa ja täyttävät niitä turhaan. Myös vanhat, moduulirakenteen ulkopuoliset ominaisuudet täytyy aina käydä läpi ja korjata viittaamaan uudistettuihin moduuleihin.

commons/helpers/common/		88.46%	46/52	83.78%	31/37
commons/helpers/date/		100%	86/86	96.55%	56/58
commons/helpers/form/		100%	41/41	100%	21/21
commons/helpers/modal/		100%	23/23	100%	10/10
commons/helpers/sorter/		87.1%	27/31	90.91%	20/22
commons/helpers/string/		100%	17/17	100%	13/13

Kuvio 36. Yksikkötestien kattavuus työn loppuvaiheessa

Tietenkään kaikki osa-alueet työstä eivät tuntuneet turhalta tai menneet hukkaan. Opinnäytetyön päätteeksi yksikkötestien kattavuus on saatu erittäin hyvälle tasolle käsiteltyjen funktioiden osalta (kuvio 36) ja työssä rakennetut *helpers*-moduulit jäävät pysyvästi käyttöön Wilman kehityksen tueksi. Myös oma osaaminen on kehittynyt huomattavasti ja yrityksessä on saatu parempi kokonaisvaltainen kuva siitä, mihin suuntaan vanhaa JavaScriptiä kannattaa kehittää.

LÄHTEET

Stack Overflow 2017. Developer Survey. Viitattu 12.7.2017.
https://insights.stackoverflow.com/survey/2017

Herron, J. 2011. Node Web Development. Packt Publishing.

Juzer, A. 2013. Instant Node Package Manager. Packt Publishing.

Jasmine 2017A. Getting Started. Viitattu 22.7.2017.
https://jasmine.github.io/pages/getting_started.html

Jasmine 2017B. Introduction. Viitattu 22.7.2017.
https://jasmine.github.io/2.5/introduction

Pillora, J. 2014. Getting Started with Grunt: The JavaScript Task Runner. Packt Publishing.

Art of Unit Testing 2011. Definition of a Unit Test. Viitattu 6.8.2017.
http://artofunittesting.com/definition-of-a-unit-test/

Simple Programmer 2010. The Purpose of Unit Testing. Viitattu 13.8.2017.
https://simpleprogrammer.com/2010/10/15/the-purpose-of-unit-testing/

InnoArchitech 2014. JavaScript, ECMA–262, TC39, and ECMAScript Transpilers Explained. Viitattu 19.8.2017.
https://www.innoarchitech.com/javascript-ecma262-tc39-ecmascript-transpilers-explained/

Felix Rieseberg 2015. A Quick Intro to the Future of JavaScript. Viitattu 19.8.2017.
https://felixrieseberg.com/ecmascript6-introduction/

Electron 2017. Build cross platform desktop apps with JavaScript, HTML, and CSS. Viitattu 26.9.2017.
https://electron.atom.io/

Stack Overflow 2015. Global Namespace. Viitattu 26.9.2017.
https://stackoverflow.com/a/13352212

Preethi Kasireddy 2016. JavaScript Modules: A Beginner’s Guide. Viitattu 27.9.2017.
https://medium.freecodecamp.org/javascript-modules-a-beginner-s-guide-783f7d7a5fcc

GitHub 2017A. JavaScript-puutarha. Viitattu 28.9.2017.
https://bonsaiden.github.io/JavaScript-Garden/fi/

2ality 2014. ECMAScript 6 modules: the final syntax. Viitattu 30.9.2017. http://2ality.com/2014/09/es6-modules-final.html

GitHub 2017B. Karma - How It Works. Viitattu 30.9.2017. http://karma-runner.github.io/1.0/intro/how-it-works.html

Microsoft 2012. Bundling and Minification. Viitattu 1.10.2017. https://docs.microsoft.com/en-us/aspnet/mvc/overview/performance/bundling-and-minification

Webpack 2017A. Core Concepts. Viitattu 1.10.2017. https://webpack.js.org/concepts/

Webpack 2017B. Commons Chunk Plugin. Viitattu 15.10.2017. https://webpack.js.org/plugins/commons-chunk-plugin/

Delphi Basics 2017. TDateTime. Viitattu 2.10.2017. http://www.delphibasics.co.uk/RTL.asp?Name=TDateTime

Mozilla Developer Network 2017. Global Object. Viitattu 8.10.2017. https://developer.mozilla.org/en-US/docs/Glossary/Global_object

Kleo Petrov 2016. Everything About Babel. Viitattu 15.10.2017. https://kleopetrov.me/2016/03/18/everything-about-babel/

GitHub 2017C. ECMAScript 6 compatibility table. Viitattu 15.10.2017. https://kangax.github.io/compat-table/es6/

Visma InSchool 2017. Opetustoimen hallinto. Viitattu 10.11.2017. https://www.visma.fi/inschool/