SAVONIA

OPINNÄYTETYÖ- **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

FRONT–END SCRIPTIEN TESTAUS

TEKIJÄ/T: Teemu Väänänen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Sähkötekniikan koulutusohjelma	
Työn tekijä(t) Teemu Väänänen	
Työn nimi Front-End Scriptien Testaus	
Päiväys 20.5.2019	Sivumäärä/Liitteet 45
Ohjaaja(t) Lehtori Jussi Koistinen, lehtori Keijo Kuosmanen	
Toimeksiantaja/Yhteistyökumppani(t) Solteq Oy	

Tiivistelmä

Opinnäytetyön tavoitteena oli perehtyä front-end JavaScript koodin testaamisen automatisointiin, ja tähän tarkoitukseen olemassa olevien työkalujen käyttämiseen sekä soveltuvuuteen kyseiseen tarkoitukseen. Työssä käsiteltyjen testaustyökalujen joukosta on vertailun perusteella tarkoitus rajata työkaluista parhaiten toimeksiantajan sovelluksen front-end testauksen automaation kehittämistä varten parhaiten soveltuvat.

Työn alussa perehdyttiin tarkemmin testaukseen yleisellä tasolla. Osiossa keskityttiin eri testauksen tyyppeihin, testaamisen automatisoimisen etuihin sekä koodin automaatiotestien eri luokituksiin. Pääkohteena oli yksikkötestaus, ja testivetoisen kehityksen käsite, mutta osiossa avattiin myös integraatio- ja funktionaalisen testaamisen käsitteitä.

Seuraavassa osiossa keskityttiin eri JavaScript-koodin testauksen automatisointia varten kehitettyihin työkaluihin. Käsitellyt työkalut valikoitiin artikkeleiden, ohjelmistokehittäjien mielipidekyselyiden, jne. perusteella. Käsitellyt työkalut esiteltiin lyhyesti sekä lopussa esitellään omat mielipiteet näiden käyttökelpoisuudesta. Työkalujen esittelyn jälkeen testikoodin rakennetta demonstroitiin yksinkertaisen esimerkkisovelluksen avulla.

Lopputuloksena koottiin ajatukset testaamisen automatisoinnista sekä työssä käsitellyistä testaamisen automatisoinnin työkaluista. Testatuista työkaluista muodostettiin raportti niiden vertailun tuloksista. Työkaluista tuotiin esille joukosta edukseen erottuneet sekä omalta osalta parhaimmaksi todettu. Jatkokehityksen aiheeksi jäi vielä front-end yksikkötestaamisen automatisoinnin integrointi toimeksiantajan koodikantaan.

Avainsanat
Yksikkötestaus, Testaus, JavaScript

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Electrical Engineering			
Author(s) Teemu Väänänen			
Title of Thesis Front-End Script Testing			
Date	20.5.2019	Pages/Appendices	45
Supervisor(s) Mr Jussi Koistinen, Senior Lecturer, Mr Keijo Kuosmanen, Senior Lecturer			
Client Organisation /Partners Solteq Oy			

Abstract

The goal of this thesis was to introduce the concept of automated testing of the front-end JavaScript code, and different frameworks and tools available for that purpose. Out of the frameworks covered in this thesis, the best suited ones would be chosen for developing an automated testing solution for the commissioner's software front-end codebase.

First the concept of automated testing of software was studied. The focus was on unit testing and the concept of test-driven development, but other categories of automated tests like integration testing, and graphical user interface testing were also studied. Next, different JavaScript automated testing frameworks were studied. The tested tools were split into two categories: JavaScript unit testing frameworks, and functional testing frameworks. Different frameworks were tested by writing test code using the specific framework. Documentation, articles, online developer surveys etc. were used to select the different frameworks tested. The basic features of the frameworks are described, with a short opinion on their usability given after. A simplistic example software was used to demonstrate what the test code is like.

Finally, as the result of this thesis, a report on automated JavaScript testing frameworks was created. Several tested frameworks were introduced, and their features looked at. The tools were compared against one another, and the most stand out ones were brought up. The actual implementation of front-end unit testing is still in progress, and the next step would be integrating a unit testing to commissioner's codebase.

Keywords
Unit testing, Testing, JavaScript

SISÄLLYSLUETTELO

MÄÄRITELMÄT

TDD-Test Driven Development

Ohjelmistokehityksen käsite, jossa koodin yksikkötestaaminen yhdistetään osaksi kehitysprosessia. Testit kirjoitetaan ennen varsinaista koodia, minkä jälkeen toteutetaan koodi siten, että testit menevät läpi. (SearchSoftwareQuality, 2019)

BDD-Behavior Driven Development

Ohjelmistokehityksen käsite, tuotettu koodi on suunniteltu loppukäyttäjän käyttäjäkokemusta painottaen. (SearchSoftwareQuality, 2019)

CLI-Command Line Interface

Tekstipohjainen käyttöliittymä, jolla voidaan ohjata tietokonetta. Esimerkiksi Windows: in komentokehote tai PowerShell-terminaali. (SearchSoftwareQuality, 2019)

GUI-Graphical User Interface

Käyttöliittymä, jonka kautta käyttäjän on mahdollista ohjata tietokonetta visuaalisten valikoiden ja kuvakkeiden kautta tekstikomentojen sijaan. Esimerkiksi Windows, Mac Os tai Android. (SearchSoftwareQuality, 2019)

API-Application Programming Interface

Koodin rajapinta, joka mahdollistaa kahden eri ohjelman välisen kommunikaation määriteltyjen (SearchSoftwareQuality, 2019)

NPM-Node Package Manager

Node.js open source pakettien jakamiseen ja kehittämiseen tarkoitettu arkisto. (Npm, 2019)

JSON-JavaScript Object Notation

JSON on web-selaimissa käytettävä helppolukuinen datan esitysformaatti. (SearchSoftwareQuality, 2019)

DOM-Document Object Model

DOM on rajapinta HTML- ja XML- dokumenttien mallintamiseen. DOM määrittää dokumentin rakenteen sekä kuinka sitä voidaan käsitellä ja muokata. (SearchSoftwareQuality, 2019)

Assertio

Assertio on testaamisessa käytettävä funktio, joka palauttaa boolean arvon, joka kertoo, toimiiko koodi oletetulla tavalla.
(Cary, 2010)

Mock-funktio

Mock on olio tai funktio, jolla korvataan jokin koodin osa-alue. Mockilla simuloidaan korvattavan olion tai funktion toimintaa halutulla tavalla.
(Paralogarajah, 2017)

1 JOHDANTO

Opinnäytetyön tilaajana on Solteq Oy. Solteq on ohjelmistoalan yritys, joka toimittaa eri IT-alan ratkaisuja. Yksi myydyistä tuotteista on energiayhtiöille tarjottava Inpulse4Utilities-web sovellus. Sovellus sisältää paljon selaimen päässä suoritettavaa JavaScript-koodia, jolle ei ennestään ole olemassa minkäänlaista automaatiotestausta. Skriptejä joudutaan aika-ajoin muokkaamaan, ja ylläpitämään, jolloin koodiin voi helposti ilmetä virheitä.

Opinnäytetyön tavoitteena on tutustua JavaScript koodin testaamisen automatisointiin sekä eri testausalustoihin, joita kyseiseen tarkoitukseen on tarjolla. Työssä avataan koodin automaattisen testaamisen käsitettä, ja selvitetään miksi sitä olisi hyvä toteuttaa. Testausalustoista käydään lyhyesti läpi niiden ominaisuudet sekä niiden toimivuus testin kirjoittamiseen. Työssä tutkittavat testausalustat ovat lähtökohtaisesti avoimen lähdekoodin työkaluja. Työssä on tarkoitus verrata eri testausalustoja toisiinsa, ja selvittää onko jokin/jotkin selkeästi toimivampia ratkaisuja testien automatisointia varten.

Työssä on tarkoitus tutustua pääsääntöisesti yksikkötestaukseen, mutta myös integraatiotestaukseen sekä verkkosivun käyttöliittymän toiminnan testaamiseen. Tutkittujen testaustyökalujen kokeilun ja vertailun perusteella olisi tarkoitus valita front-end JavaScript-koodin testaamiseen parhaiten soveltuvat työkalut. Jatkokehityksenä työn tuloksien perusteella on testaamisen automatisointiratkaisun kehittäminen valituilla alustoilla.

2 TEORIA

Tässä kappaleessa käydään lyhyesti läpi aiheeseen liittyvää teoriaa. Käsiteltävä teoria koostuu JavaScript-ohjelmointikielestä sekä koodin testaamisesta, ja testaamisen eduista.

2.1 JavaScript

JavaScript on johtava nykyaikaisessa web-kehityksessä käytettävä client-puolen skriptauskieli. JavaScriptiä hyödynnetään useissa eri käyttökohteissa verkkosivujen lomakkeiden validoinnista monimutkaisten käyttöliittymien luomiseen. JavaScript on HTML: n ja CSS: n lisäksi kolmas verkkosivuilla käytetyistä kielistä, ja se vastaa client-puolen toiminnallisuudesta ja ohjelmalogiikasta. Tästä johtuen verkko-ohjelmistojen kehittäjien olisi syytä opetella se hyvin. (Powel & Schneider 2012, 3)

Ongelmana on kuitenkin eri selainten erilaiset tavat implementoida JavaScript-metodeja, mistä johtuen JavaScript: n käyttö voi olla haasteellista. (Powel & Schneider 2012, 26) JavaScript kuitenkin kehittyy nopeaa tahtia, ja monet eri työkalut ovat tehneet JavaScript-kehittämisestä helpompaa. Node package manager (npm) automatisoi JavaScript-kirjastojen lataamista ja päivitystä. Webpack ja vastaavat työkalut mahdollistavat JavaScript-moduulien paketoinnin. Babel-kääntäjä puolestaan helpottaa koodin tuottamista muuntamalla uusimpia JavaScript-ominaisuuksia käyttäviä projekteja selaimien tukemaan muotoon. (Jang, 2017)

2.1.1 JavaScript-versiot

Brendan Eich kehitti JavaScript-kielen vuonna 1995, ja myöhemmin 1997 siitä tuli ECMA-standardi. Standardin virallinen nimi on ECMA-262 ja kielen virallinen nimi on ECMAScript. ECMAScriptistä on kehitetty useita versioita, joista viimeisin on kesäkuussa 2017 julkaistu versio 8. (W3Schools, 2018 & ECMA-262 Standard, 2018) Nykyiset selaimet eivät kuitenkaan täysin tue uusimpia standardeja. ECMAScript 5 on tuettuna, kuten myös ECMAScript 6, Internet Exploreria lukuun ottamatta. Olettaen että käytetään selainten uusimpia versioita. Selaimet kuitenkin implementoivat uusien standardien lisäämiä ominaisuuksia ajan mittaan. (ECMAScript Compatibility Table)

2.1.2 Node.js

Node.js on Chrome: n V8-JavaScript-moottorilla rakennettu asynkroninen, tapahtumavetoinen JavaScript-ympäristö. Node.js on suunniteltu kasvavien verkkosovellusten kehittämiseen. (Node, 2018)

Node.js o tarkoitettu JavaScript-koodin suorittamisen selaimen ulkopuolella, kuten suoraan tietokoneella tai serverillä. (MDN Web docs, 2018)

2.1.3 Node package manager

Node package manager (npm) on työkalu Node.js: n pakettien / moduulien hallintaa varten. Npm sisältää cli-rajapinnan, joka mahdollistaa npm-pakettien lisäämisen ja hallinnoinnin Node.js-projektissa. (Npm, 2018)

2.1.4 TypeScript

TypeScript on JavaScriptin ylijoukko, joka lisää muuttujatyyppejä, luokkia ja moduuleja JavaScript: iin. TypeScript hyödyntää samaa syntaksia kuin tavallinen JavaScriptkin, ja sitä voidaan käyttää yhdessä tavallisen JavaScriptin kanssa. TypeScriptin kanssa on myös mahdollista käyttää kaikkia JavaScript kirjastoaja. TypeScript ei kuitenkaan toimi selaimessa sellaisenaan, vaan se käännetään ennen suorittamista tavalliseksi JavaScriptiksi. (TypeScript, 2018)

Jos web-sovelluksen back-end puolella on käytössä oliopohjainen ohjelmointikieli, kuten c#, TypeScript mahdollistaa front-end-koodin kirjoittamisen tätä lähemmin vastaavammassa muodossa, ja siten voi tehdä selaimen- ja serverin pään koodista toiseen siirtymisestä mutkattomampaa. Toisaalta TypeScriptin staattiset tyypitykset myös jossain määrin rajoittavat, miten JavaScript-koodia tulee kirjoittaa.

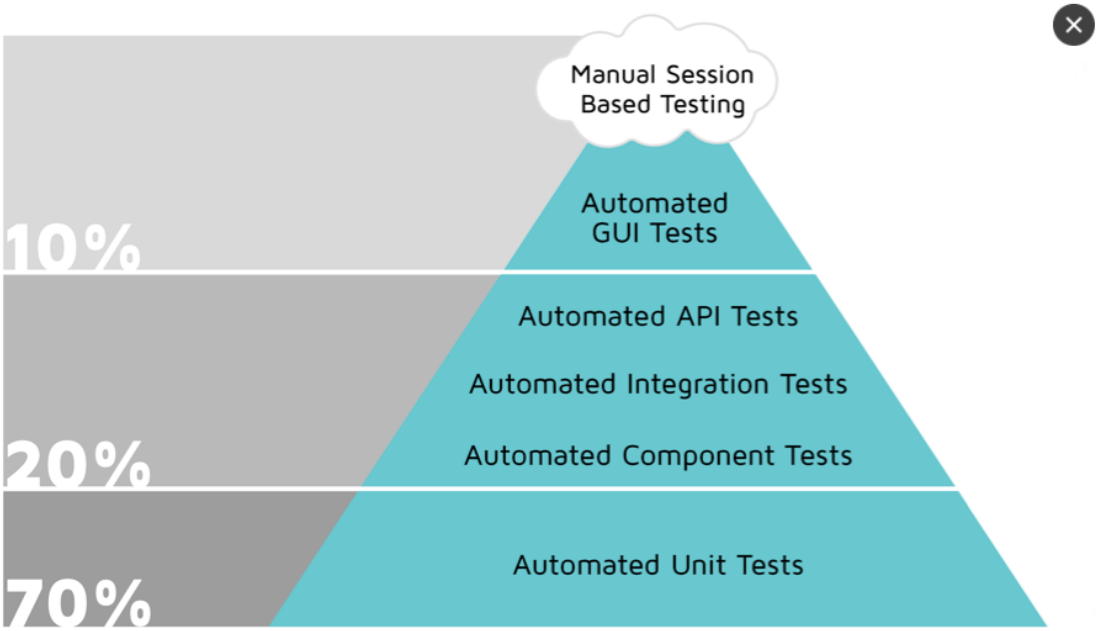
2.2 Koodin testaaminen

Testaamisella havaitaan ongelmia ja arvioidaan koodin laatua. Testausta tehdään, koska virheiden mahdollisuus tiedostetaan. Sovellusten testaamisen ydin on testien määrittäminen halutulle testikohteelle. Testitapaus sisältää testikohteen tunnuksen, lyhyen kuvauksen testin tarkoituksesta ja vaatimuksista, syötöstä sekä odotetusta tuloksesta. (Jorgensen, 2016, Chapter 1)

Testauksella varmistetaan, että ohjelman logiikka toimii oikein, eli tietyllä syötöllä saadaan ohjelmasta aina palautettua sama tulos. (Eugene, 2010, Chapter 1)

2.3 Testityypit

Testejä voidaan luokitella useaan eri tyyppiin, mutta web-kehityksen kannalta oleellisimpia ovat: yksikkötestit, integraatiotestit ja funktionaaliset testit (Zaidman, 2018).



Kuva 1 Automatisoidun testauksen pyramidi (Bushev, 2017)

Kuvassa 1 on havainnollistettu automatisoitujen testien jako. Pohjalla olevat yksikkötestit ovat alimman tason testejä, joiden käytöllä tulisi olla suurin prioriteetti. Alemman tason testit ovat nopeampia suorittaa, ja siten myös niistä saadaan nopeammin palautetta. Alemman tason testit ovat myös stabiilimpia. Kaikki testaus olisi siis hyvä suorittaa alimmalla tasolla, millä se on mahdollista. Rajaamalla testien kohteet mahdollisimman pieniin osa-alueisiin on niiden suorittaminen helpompaa, nopeampaa ja testikoodin ylläpitäminen vaivattomampaa. (Cohn, 2009 & Bushev, 2017)

2.3.1 Yksikkötestaus

Yksikkötestauksella viitataan yhden koodin osan, kuten luokan tai funktion toiminnallisuuden testaamiseen. Yksikkötesteillä varmistetaan, että yksittäiset komponentit toimivat eristettyinä toisistaan. Yksikkötestien tulisi olla yksinkertaisia, nopeita sekä toimia hyvinä virheraportteina. (Elliot 2016)

Yksikkötestit ovat automatisoidun testauksen tärkein osa-alue. Yksikkötestit ovat erinomaisia, koska niiden avulla ohjelmoija saa tarkkaa tietoa havaituista virheistä. Yksikkötestit kirjoitetaan useimmiten samalla ohjelmointikielellä, kuin ohjelman varsinainen koodi, joten ne ovat miellyttäviä testejä kirjoittaa ja huoltaa. (Cohn, 2009)

2.3.1.1 Yksikkötestien suorittamisen kannattavuus

Testien kirjoittaminen ei ole helppoa. Automaattisten testien kirjoittaminen vaatii ylimääräistä työtä, ja varsinaista koodia voidaan myös joutua muokkaamaan yksikkötestien mahdollistamiseksi. Testausta suunniteltaessa ja tehdessä voi tulla kyseenalaistettua sen kannattavuutta. Jos testien tekeminen olisi helppoa, ei niiden arvokkuutta olisi syytä kyseenalaistaa. Onkin hyvä nostaa esille, mikä yksikkötestien kirjoittamisesta tekee haasteellista. (Sonmez, 2010)

Itse testien kirjoittaminen on rakenteeltaan helppoa, ainakin kun sen oppii. Ideaalisessa tilanteessa testattavalla koodilla ei ole mitään ulkoisia riippuvuuksia, jolloin testit voidaan luoda ilman ongelmia. Varsinainen testattava koodi harvoin kuitenkaan on rakenteeltaan yksinkertainen. Testit vaikeutuvat, jos koodi on esimerkiksi tilariippuvainen. Testit eivät edelleenkään ole välttämättä monimutkaisia, mutta testikoodissa joudutaan nyt hoitamaan myös tilan hallinta. Ideaalisessa koodissa yksikkötesti voidaan kohdistaa yksittäiseen funktioon. Jos funktion toiminta on riippuvainen tilasta, joudutaan testiin lisäämään elementtejä myös funktion ulkopuolelta. (Sonmez, 2010)

Tilan hallinta ei erityisesti vaikeuta testien kirjoittamista, vaan lähinnä lisää työmäärää. Testauksen vaikeus tulee esille, kun testattavaan koodiin on liitetty ulkoisia riippuvuuksia. Testattava funktio voi esimerkiksi lukea tai tallentaa dataa tietokantaan. Koska kyseessä on yksikkötesti, ei varsinaista tietokantaa oteta testaukseen mukaan, vaan se korvataan esimerkiksi mock-funktiolla. (Sonmez, 2010)

Testattava voi koodi voi sisältää myös molemmat: ulkoisen riippuvuuden sekä tilan hallinnan, jolloin myös testeissä joudutaan huomioimaan molemmat. Mitä enemmän riippuvuuksia testattava koodi sisältää, sitä monimutkaisemmaksi testikoodi joudutaan kirjoittamaan. Mitä monimutkaisempaa testikoodi on, sitä suuremmalla todennäköisyydellä virhe löytyykin testistä. Oleellista onkin pohtia, miten pitkälle yksikkötestejä on järkevää kirjoittaa. Yksikkötestien tulisi testata koodin logiikkaa, ei implementaatiota. Jos testikoodi on monimutkaisempaa kuin testattava koodi, ei yksikkötestien tekeminen todennäköisesti ole vaivan arvoista. (Sonmez, 2010)

2.3.1.2 Yksikkötestauksen edut

Testit ovat ensimmäinen ja paras suoja ohjelman virheitä vastaan. Yksikkötestit olisi hyvä kirjoittaa ennen varsinaista koodia, jolloin ne toimivat ohjeena varsinaista toteutusta tehtäessä. Testin tulisi tarjota selkeä kuvaus testattavasta toiminnosta sekä toimia virheraporttina niiden epäonnistuessa. (Elliot, 2015)

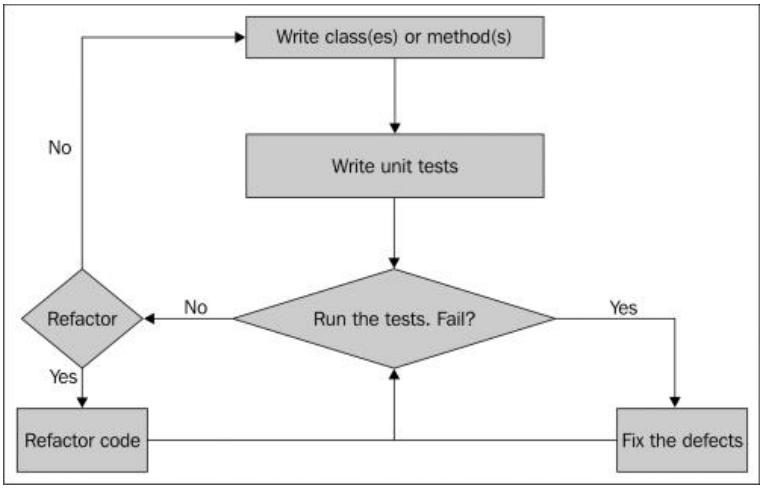
Koodia joudutaan testaamaan joka tapauksessa, ja jos prosessia ei ole automatisoitu, joudutaan se tekemään manuaalisesti. Pienemmissä projekteissa ei varsinaisesti ole tarvetta automatisoidulle testaukselle, mutta projektien koon kasvaessa testaukseen kuluu enemmän ja enemmän aikaa, ja on yhä todennäköisempää, että jokin koodin osa jää testaamatta. Yksikkötestauksen implementoinnilla saadaan vähennettyä työmäärää, ja palaute saadaan nopeammin. Yksikkötestit voidaan ajaa yhdellä komennolla, tai automatisoida testien suorittaminen aina, kun koodiin tehdään muutoksia, jolloin muutokset voidaan tehdä ilman pelkoa, että jokin muu koodin osa ei enää toimi. (Tang, 2016)

Yksikkötestauksen implementointi, etenkin TDD-mallin noudattaminen, johtaa modulaarisemman ja helpommin luettavan koodin tuottamiseen. Tämä johtaa funktioihin ja luokkiin, jotka tekevät

yksittäisiä asioita erittäin hyvin. Testifunktiot toimivat myös hyvinä kuvauksina ja esimerkkeinä, siitä mitä testattavien funktioiden ja luokkien tulisi tehdä. (Tang, 2016)

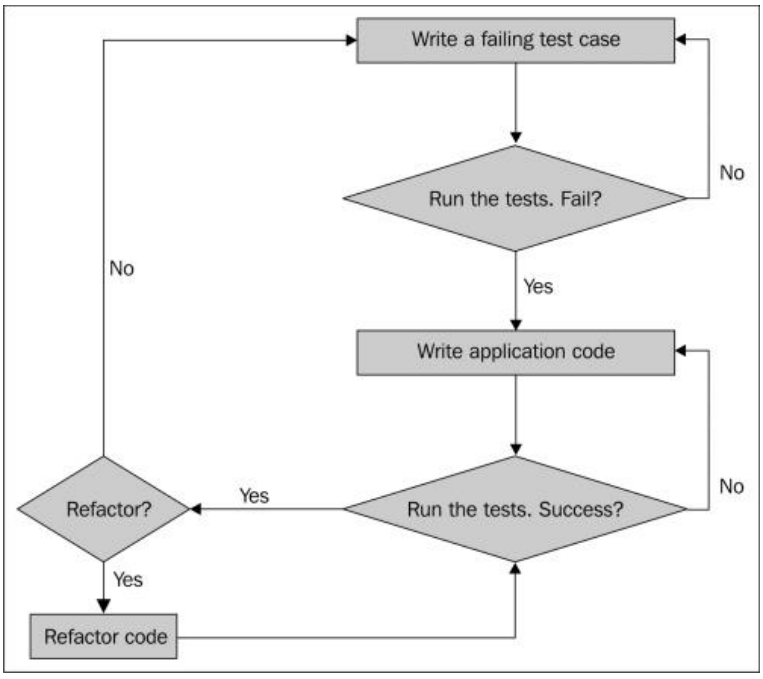
2.3.1.3 Testivetoinen kehitys

Yksikkötestien luomista varten on olemassa kaksi yleisesti tunnettua lähestymistapaa. ”Traditional unit testing”, eli perinteinen yksikkötestaus sekä ”Test Driven Development” (TTD), eli testivetoinen kehitys. Perinteisen yksikkötestauksen mallin mukaisesti kirjoitetaan sovelluksen koodi ensin, minkä jälkeen on testikoodin vuoro. Valmis testikoodi ajetaan, ja jos testi ei mene läpi, korjataan sovelluksen koodista löytyvä virheet. (Saleh 2013, Chapter 1)



Kuva 2 Traditional Unit Test (Saleh 2013, Chapter 1)

Testijohtoisessa mallissa (TDD) on perinteisen testausmallin järjestys käänteinen. TDD-prosessissa kirjoitetaan testit ennen varsinaista koodia. Kirjoitetaan testi, varmistetaan että se ei mene läpi, minkä jälkeen kirjoitetaan varsinainen koodi siten, että se läpäisee testin ja muokataan koodia tarvittaessa. (Elliot, 2016)



Kuva 3 Test Driven Development (Saleh 2013, Chapter 1)

2.3.2 Integraatiotestaus

Integraatiotestauksella varmistetaan, että eri yksiköt toimivat keskenään halutulla tavalla. Integraatiotestauksella testataan sovelluksen toimintaa erillään sen käyttöliittymästä. (Elliot 2016, Cohn 2009) Integraatiotestauksella on mahdollista havaita odottamattomia virheitä, joissa yhden asian korjaaminen rikkoo jonkin toisen. On myös huomioitava, että todellisissa sovelluksissa eivät kaikki yksiköt ole puhtaita tai testattavissa. Joitakin koodin osa-alueita voi olla mahdollista testata vain osana suurempaa kokonaisuutta. (Zaidman, 2018)

2.3.3 Funktionaalinen testaus

Funktionaalisilla testeillä varmistetaan, että sovellus toimii niin kuin sen tulisi toimia käyttäjän näkökulmasta. Funktionaalisissa testeissä syötetään data UI: lle, ja varmistetaan, että sovellus vastaa kuten sen pitäisi (Elliot 2016). Automaattista käyttöliittymän testausta tulisi tehdä mahdollisimman vähän, sillä käyttöliittymän testit ovat hauraita ja vievät paljon aikaa (Cohn, 2009).

2.4 JavaScript yksikkötestaus

Tyypilliselle JavaScript-koodille voi olla vaikeata kirjoittaa yksikkötestejä. Tähän yhtenä syynä on, ettei selaimelle kirjoitettua JavaScript-koodia aina ole helppoa jakaa selkeisiin testattaviin yksikköihin. Front-end koodi on yleensä kirjoitettu juuri sitä käyttävää sivua varten, ja on siten myös laajalti sidoksissa kyseisen sivun DOM: iin sekä back-endin logiikkaan. JavaScript-koodi voi olla jopa suoraan osa sivun html toteutusta. Tämä koskee etenkin puhdasta JavaScript-koodia, johon ei ole käytetty mitään erillistä kirjastoa. (Zaefferer 2012)

Koodin erottaminen omaan tiedostoonsa sekä erillisten DOM: n abstraktoivien kirjastojen / rajapintojen käyttäminen tekee testaamisesta helpompaa, joskin edelleen haasteellista. Jos koodi kirjoitetaan paremmin testattavaan muotoon, huomattaisiin, että testien kirjoittamisesta tehdään paljon helpompaa. Samalla koodi saadaan myös parempaan muotoon, jolloin myös sen huoltaminen, debuggaaminen, ja jatkokehittäminen on helpompaa. (Zaefferer 2012)

Kuvan 4 esimerkkikoodissa on testaamisen kannalta neljä vaikeuttavaa asiaa:

- Rakenteen puute. Suurin osa koodista suoritetaan $(document).ready()-callback- funktiossa sekä nimeämättömissä funktioissa, joihin ei ole mahdollista päästä yksikkötesteissä käsiksi.
- Liian monimutkaiset funktiot. Yksittäiset funktiot suorittavat liian monta toimintoa.
- Piilotettu tai jaettu tila. Ei esimerkiksi ole mahdollista tarkistaa asetetaanko pending-muuttujan arvo oikein.
- Liika liitännäisyys. Ajax-funktion ei tarvitsisi olla yhdistettynä DOM: iin. (Murphey, 2013)

Kuvan 4 koodia voidaan parantaa jakamalla se muutamaan selkeään osa-alueeseen. Ongelmana koodissa on, että eri osa-alueiden toimenpiteitä suoritetaan sekaisin ympäri koko koodia. Koska

yksikään koodin osa ei keskity suorittamaan selkeästi mitään yksittäistä toimenpidettä, ei sille ole mahdollista kirjoittaa kunnollista yksikkötestiä. (Murphey, 2013)

```
var tmplCache = {};

function loadTemplate (name) {
  if (!tmplCache[name]) {
    tmplCache[name] = $.get('/templates/' + name);
  }
  return tmplCache[name];
}

$(function () {

  var resultsList = $('#results');
  var liked = $('#liked');
  var pending = false;

  $('#searchForm').on('submit', function (e) {
    e.preventDefault();

    if (pending) { return; }

    var form = $(this);
    var query = $.trim( form.find('input[name="q"]').val() );

    if (!query) { return; }

    pending = true;

    $.ajax('/data/search.json', {
      data : { q: query },
      dataType : 'json',
      success : function (data) {
        loadTemplate('people-detailed.tmpl').then(function (t) {
          var tmpl = _.template(t);
          resultsList.html( tmpl({ people : data.results }) );
          pending = false;
        });
      }
    });

    $('<li>', {
      'class' : 'pending',
      html : 'Searching &hellip;'
    }).appendTo( resultsList.empty() );
  });

  resultsList.on('click', '.like', function (e) {
    e.preventDefault();
    var name = $(this).closest('li').find('h2').text();
    liked.find('.no-results').remove();
    $('<li>', { text: name }).appendTo(liked);
  });

});
```

Kuva 4 Esimerkki tyypillisestä selaimen JavaScript-koodista (Murphey, 2013)

3 YKSIKKÖTESTAUSTYÖKALUT

Tässä kappaleessa käydään läpi yksikkötestausta JavaScript-koodin osalta sekä eri avoimen lähdekoodin työkaluja, joilla yksikkötestejä on mahdollista toteuttaa.

3.1 Jasmine

Jasmine on BDD testausalusta JavaScript-koodille. Jasmine ei ole riippuvainen selaimista, DOM: sta, tai mistään JavaScript sovelluskehyksistä. Jasmine tarjoaa yksinkertaisen ja helppokäyttöisen syntaksin, joten testien kirjoittaminen on yksinkertaista. Testaus on mahdollista sekä selaimessa, että Node.js: ssä. (Jasmine, 2018)

Jasmine on yksi vanhemmista JavaScript-yksikkötestausta varten kehitetyistä työkaluista. Se on myös yksi eniten käytetyimmistä työkaluista. Jasmine testaustyökalun tärkeimpiä ominaisuuksia ovat

- Sisältää valmiina kaiken testaamista varten tarvittavan, joten testien kirjoittamista varten ei tarvita esimerkiksi erillistä assertio-kirjastoa.
- Testiympäristö assertio-funktioineen asetetaan globaaleiksi arvoiksi. Ajettaessa testejä Node.js ympäristössä, ei jokaiselle testitiedostolle tarvitse kertoa kirjastoa käytettäväksi.
- On ollut olemassa vuodesta 2009 lähtien, joten siihen liittyen on tarjolla paljon materiaalia, kuten artikkeleita ja siihen perustuvia työkaluja ja lisäosia. (Zaidman, 2018)
- Testien suorittaminen joko selaimessa. Testien suorittaminen Node.js: n kautta on usein nopeampaa, mutta front-end koodista ei kaikkea voida testata ilman oikeaa selaimen ympäristöä.

3.2 Mocha

Mocha on JavaScript-yksikkötestaukseen kehitetty työkalu, jolla testejä voidaan suorittaa sekä selaimessa, että Node.js: llä. Mocha testit suoritetaan sarjamuotoisesti, mahdollistaen joustavan ja tarkan raportoinnin. (Mocha, 2018) Mocha on tällä hetkellä käytetyin testauskirjasto (Zaidman, 2018).

Toisin kuin edellä esitelty Jasmine, Mocha on pelkkä rajapinta testien suorittamista varten. Tämä tarkoittaa, että se sisältää pelkän rakennemallin testien kirjoittamista varten, ja siten tarvitsee erillisen assertiokirjaston. Tämä ei kuitenkaan ole valinnan osalta erityisen merkityksellistä, sillä erillisen kirjaston lisäys ja hallinnointi on helppoa npm: n avulla.

Mochan tärkeimpiä ominaisuuksia:

- Testiympäristöä varten luodaan globaalit arvot. Testikirjastoa ei siten tarvitse erikseen lisätä testitiedostoihin.
- Ei sisällä omia assertioita. Käytetty kirjasto on erikseen valittavissa.
- Useita liitännäisiä, eri testitilanteita varten. Mochan ominaisuuksien laajentaminen on mahdollista useiden eri open-source lisäpakettien kautta. (Zaidman, 2018)

- Watch-mode testitiedostojen muutoksien seuraamista varten. Testien suorittaminen mahdollista konfiguroida automaattisesti suoritettaviksi tiedostoihin tehtyjen muutosten yhteydessä.

3.3 Jest

Jest on Facebookin kehittämä, ensisijaisesti React-applikaatioiden testaamiseen, mutta soveltuu hyvin muidenkin JavaScript-sovellusrajapintojen, kuten Vue.js, AngularJs, Angular, MobX, ja Reduxin testaamiseen. Jest pohjautuu Jasmineen, mutta nykyään lähes kaikki Jasminen funktionaalisuus on korvattu, ja uusia ominaisuuksia on lisätty. (Zaidman, 2018, Jest, 2018)

Jest ajaa testit rinnakkaisesti suorituskyvyn parantamiseksi. Konsoliviestit tulostetaan yhdessä testitulosten kanssa. Eristetyt testitiedostot ja globaalin tilan resetointi varmistavat, etteivät testit joudu ristiriitaan toistensa kanssa. (Zaidman, 2018, Jest, 2018)

Jestin tärkeimmät ominaisuudet

- Sisältää kaiken testaamista varten tarvittavan: testiympäristön, assertiot, mock-funktiot.
- Koodin kattavuuden raportointi Istanbul: iin perustuvan työkalun avulla. Ei tarvita erillistä moduulia testien kattavuuden raportointia varten.
- Testiympäristö kaikkine ominaisuuksineen lisätään globaaleiksi muuttujiksi, jolloin tätä ei tarvitse tehdä erikseen testikoodissa.
- Snapshot-testaus. Mahdollistaa UI: n testaamisen odottamattomien muutosten varalta.
- Luotettavuus: Vakaantui vuoden 2017 aikana, ja pidetään nyt luotettavana frameworkinä. Jest on edelleen aktiivisen kehityksen alla, ja uusia ominaisuuksia on edelleen työn alla.
- Jestiä pidetään nopeana isoissa projekteissa, sen testien rinnakkaissuorittamisen ansiosta.
- Jest päivittää vain muokatut tiedostot, joten testien uudelleenajaminen on nopeaa watch-mode-toimintoa käytettäessä. Yksittäisen testin suorittaminen on nopeampaa muutoksia tehtäessä. (Jest, 2018 & Zaidman, 2018)

3.4 Ava

Minimalistinen JavaScript-testauskirjasto, joka mahdollistaa yksikkötestien ajamisen rinnakkaisesti (Zaidman, 2018). JavaScript on rakenteeltaan yksisäikeinen, mutta Node.js: n IO kykenee suorittamaan koodia rinnakkaisesti sen asynkronisen luonteen ansiosta. AVA hyödyntää tätä testien rinnakkaiseen ajamiseen. Kaikki testitiedostot ajetaan omina prosesseinaan, joten jokainen testitiedosto suoritetaan omassa eristetyssä ympäristössään. Tällä pakotetaan testit riippumattomiksi globaalista tilasta ja muista testeistä. (AVA, 2018)

Ava on pienikokoinen yksikkötestausalusta, joka mahdollistaa kuitenkin melko laajan konfiguraation asettamisen. Testitiedostot ajetaan myös eristyksissä toisistaan, mikä mahdollistaa globaalin tilan muuttamisen testeissä vaikuttamatta muiden testitiedostojen globaaliin tilaan.
Avan tärkeimmät ominaisuudet:

- Sisältää testiympäristön ja assertiot testien kirjoittamista varten. Testien kirjoittamiseen on käytettävä mukana tulevaa kirjastoa.
- ES6-syntaksi. Kaikki testit käännetään Babelin kautta, joten testikoodia voidaan kirjoittaa uusimmalla JavaScript-syntaksilla. Tämä on kuitenkin yksi merkittävästi testien suoritusaikaa hidastava ominaisuus.
- Ei lisää globaaleja muuttujia. Testiympäristö on lisättävä erikseen testikoodiin.
- Yksinkertainen struktuuri ja assertio-funktiot, mutta sisältää silti useita ominaisuuksia, kuten tuen asynkroniselle JavaScript-koodille sekä Promise- ja Observable-syntakseille.
- Watch-mode päivittää vain tiedostot, joihin tehtiin muutoksia, mikä nopeuttaa testien uudelleenajoon kuluvaa aikaa.
- Testien suoritetaan rinnakkain erillisissä Node-prosesseissa. Testien suorittaminen rinnakkain mahdollistaa useamman testi ajamisen samanaikaisesti, jolloin testit saadaan nopeammin suoritettua.
- Snapshot testaus käyttöliittymän muutosten varalta. (Zaidman, 2018 & AVA, 2018)

3.5 Tape

Kaikista testauskirjastoista yksinkertaisin (Zaidman, 2018). TAP-raporttia tuottava testialusta Node.js: lle ja selaimille (Tape, 2018).

Ominaisuudet

- Minimaalinen rakenne ja assertiot. Ei sisällä mitään ylimääräistä. Kirjasto on kooltaan pienikokoinen.
- Ei lisää mitään globaaleja muuttujia. Tape on lisättävä erikseen jokaiseen testitiedostoon.
- Ei jaettua tilaa testien välillä. Tape ei suosi beforeAll()-tyyppisten funktioiden käyttöä.
- Ei testitiedostojen seuraamista, ja automaattisata suoritusta. Testit täytyy suorittaa manuaalisesti koodin muutosten jälkeen. (Zaidman, 2018)

3.6 Lab

Lab on hapi.js: n kehittämä yksinkertainen testausalusta Node.js: ään. Lab on Mochan inspiroima testausalusta, joka mahdollistaa testien kirjoittamisen joko BDD-mallin mukaisesti, tai ilman. Labin tavoitteena on pitää testien suoritusmoottori mahdollisimman yksinkertaisena. Käytettäessä yhdessä hapijs: n code-assertion kirjaston kanssa, lab osaa myös ilmoittaa virheellisten tai puutteellisten assertio-funktioiden käytöstä. (Lab, 2018)

Labin tärkeimpiä ominaisuuksia

- Ei globaaleja. Testifunktioita ei aseteta Noden globaaliin tilaan, joten testikirjasto on erikseen ladattava testitiedostossa.
- Valinta pelkkien testien, ja TDD/BDD-mallin välillä.
- Tuki async/await ja Promise()-syntakseille, mikä helpottaa asynkronisen koodin testaamista.
- Tuki TypeScript: lle. TypeScript-koodia ei tarvitse erikseen kääntää testien suorittamista varten.
- Koodin testien kattavuuden raportointi sisältyy pakettiin. Testien kattavuuden raportointi.

- Ei testitiedostojen automaattista seurausta. Testien suorittaminen joudutaan tekemään manuaalisesti muutosten jälkeen.

3.7 QUnit

QUnit on tehokas, helppokäyttöinen JavaScript testausalusta. QUnit on JQueryn, JQuery UI: n, ja JQuery Mobilen käyttämä testausalusta. QUnitin kehitti John Resig JQueryn osana. Vuonna 2008 se sai oman nimen ja API: n dokumentaation, jolloin testien kirjoittaminen QUnitilla tuli mahdolliseksi muillekin. QUnit oli alun perin riippuvainen JQuerystä, kunnes se korjattiin 2009, tehden QUnitista oman standalone-työkalun. QUnit toimi alun perin vain selainten kanssa, mutta nykyisellä versiolla testien ajaminen on kuitenkin mahdollista myös Node.js ympäristössä. Nykyiselläkin versiolla QUnit kuitenkin sopii paremmin testien suorittamiseen selaimessa. (QUnit, 2018)

QUnitin oleellisimpia ominaisuuksia

- Selkeä raportointi selaimessa. Soveltuu parhaiten selaimessa käytettäväksi. Vaikka tuki Node.js: ssä testaamiselle on olemassa, ovat muut työkalut parempia vaihtoehtoja Node ympäristöön.
- Ei tukea ulkoisille assertio-kirjastoille.
- Alun perin suunniteltu JQueryn testaamiseen. Soveltuu hyvin vanhemman JQuery kirjastoa laajemmin käytävän koodin testaamiseen.
- Ei testitiedostojen seurausta ja automaattista suorittamista. Testien uudelleen suorittaminen on siten hidasta muutosten jälkeen.

3.8 Karma

Karma on Angularin kehitystiimin kehittämä työkalu testien suorittamista varten. Karma mahdollistaa testien ajamisen selaimessa sekä selainta vastaavissa ympäristöissä, kuten PhantomJS ja jsdom (Zaidman, 2018).

Karma on yksinkertainen työkalu, joka mahdollistaa JavaScript-koodin suorittamisen useissa eri selaimissa. Karma kykenee seuraamaan projektin tiedostoja ja suorittamaan testit uudestaan automaattisesti muutosten jälkeen. Karma: n tavoitteena on tehdä testien kirjoittamisesta ja suorittamisesta helppoa. (Karma, 2018)

Karma mahdollistaa yksikkötestien ajamisen selainympäristössä komentokonsolin kautta. Testit voidaan myös suorittaa useassa eri selaimessa samanaikaisesti. Karma on hyödyllinen työkalu, jos halutaan varmistaa, että testattava koodi toimii kaikissa tarvittavissa selaimissa. Karma tukee Jasminea Mochaa ja QUnit: ia, joista Jasmine on oletuksena käytettävä framework. (Karma, 2018)

Ominaisuudet

- Karma tukee Chrome, Chrome Canary, Firefox, Safari, PhantomJS, jsdom, Opera, ja Internet Explorer-selaimia. Koodi voidaan todeta toimivaksi usealla eri selaimella.

- Vain testien selaimessa suorittamista varten, joten tarvitsee erillisen yksikkötestaustyökalun testikoodin kirjoittamista varten. Käytettävä framework tarvitsee myös erillisen adapterin toimiakseen Karman kanssa.
- Testien automaattinen uudelleenlautas kaikissa määritellyissä selaimissa. Testit suoritetaan koodin muutosten yhteydessä automaattisesti.

3.9 Yksikkötestaustalustojen vertailu

Kaikkien käsiteltyjen testausalustojen avulla on mahdollista toteuttaa yksikkötestien lisääminen JavaScript-projekteihin. Kaiken kaikkiaan ero eri työkalujen välillä ei ole valtava. Oleellisempaa on valita jokin, ja alkaa yksikkötestien kirjoittaminen. Yhdenkin testausalustan käytön opettelu tekee muihin perehtymisestä helpompaa. Yksikkötestaustyökaluilla on kuitenkin joitain eroja, jotka voivat vaikuttaa sopivimman alustan valitsemiseen.

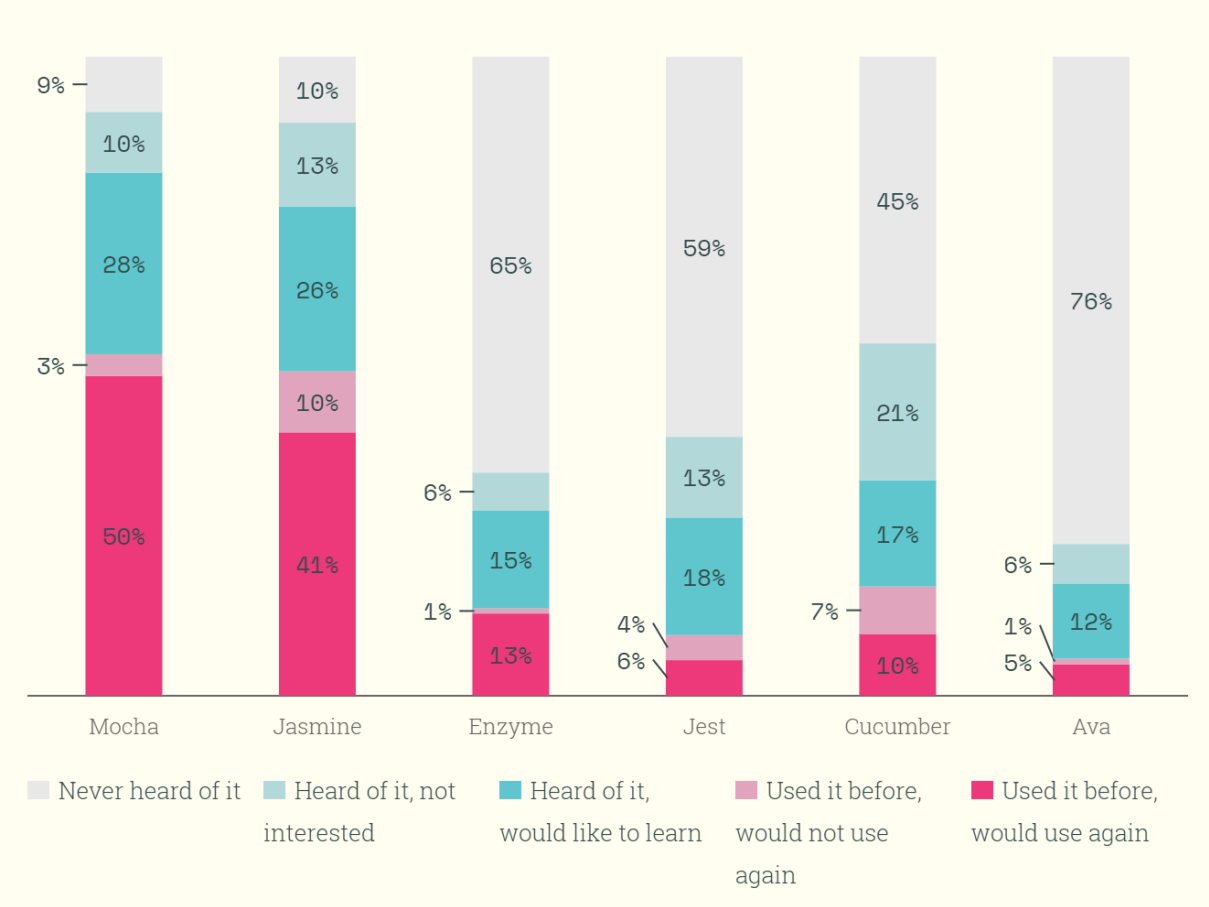
Jasmine, Mocha ja Jest tarjoavat lähes samanlaisen BDD/TDD-testirakenteen, jossa testit ryhmitetään suite-rakenteisiin. Kaikki kolme myös lisäävät omat testaukseen tarvittavat muuttujansa ja funktionsa globaaleiksi arvoiksi, jolloin kaikki testien kirjoittamista varten tarvittavat funktiot ovat saatavilla heti valitun alustan projektiin lisäämisen jälkeen. Jasmine on näistä kolmesta vanhin, ja testatuin, mutta Mocha ja Jest ovat myös korkeassa suosiossa. Mochan etuna on sen laajennettavuus. Jest on ominaisuuksiltaan käytännössä laajempi versio Jasminesta. Koska Jasmine, Jest, ja Mocha ovat syntaksiltaan lähes samanlaisia, on testausalustasta toiseen siirtyminen helppoa.

Tape on suunniteltu minimaaliseksi testaustyökaluksi, josta myöskin AVA on saanut oman inspiraationsa. Tape tarjoaa testaamiseen tarvittavan perustoiminnallisuuden, johon AVA on lisännyt oman lisätoiminnallisuutensa, kuten before- ja after- funktiot, ja selkeämmän raportoinnin testien tuloksista. AVA sisältää myös snapshot-testaamisen sekä testien rinnakkaisen suorittamisen, mikä ainakin suuremmissa projekteissa vähentää testien suorittamiseen kuluvaa aikaa.

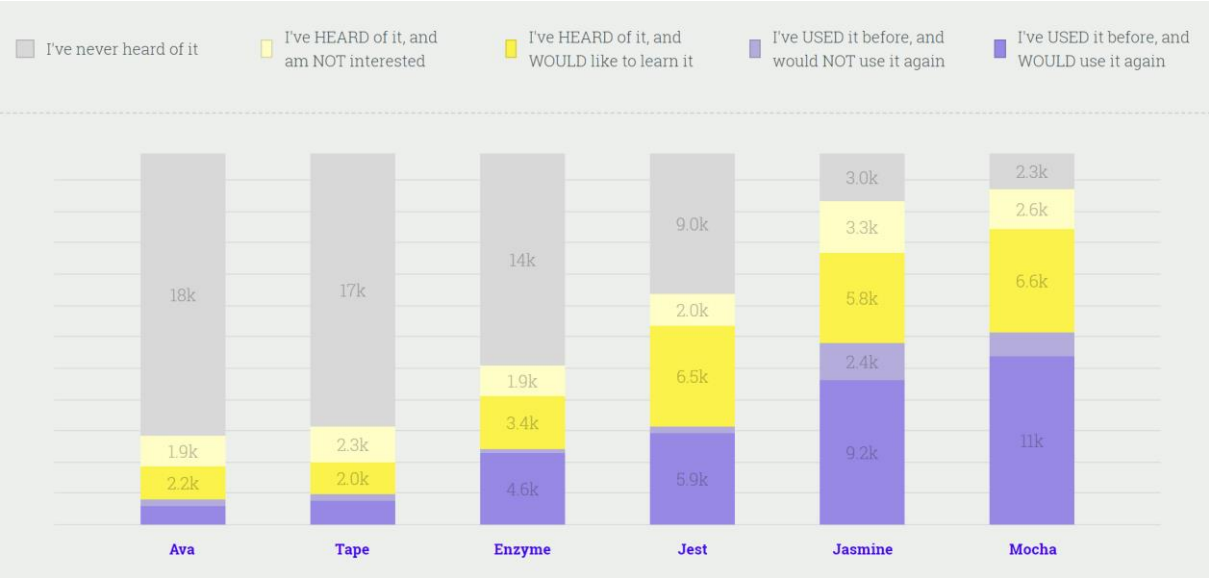
QUnit on Jasminen kanssa yksi vanhimmista JavaScript-testaustyökaluista. QUnit on toimiva valinta, joka soveltuu hyvin testien suorittamiseen selaimessa, mutta ei kuitenkaan tarjoa mitään erityisemmin joukosta erottavaa ominaisuutta. Lab ei ole erityisen tunnettu testausalusta. Labin suoritusnopeus testien, etenkin asynkronisten testien osalta on huomion arvoista, mutta muuten se ei erityisemmin erotu joukosta (Potapov, 2017).

State of JavaScript on 2016 alkaen suoritettu kysely eri JavaScript kirjastojen ja työkalujen tilasta ja käytettävyydestä, johon vastasi yli 9000 JavaScript-kehittäjää. Kysely suoritettiin uudelleen vuonna 2017, jolloin kyselyyn vastanneiden määrä oli yli 20 000 henkilöä. Yhtenä kyselyn osa-alueista oli JavaScript-testaus, ja testausalustat (Greif, 2016 & Benitte, Greif, Rambeau. 2017) Kuvissa 5 ja 6 on esiteltynä tulokset kyselyihin osallistuneiden kehittäjien kokemuksista JavaScript-testausalustoista. Jasmine ja Mocha ovat alustoista selkeästi tunnetuimmat, kun taas muut ovat vielä laajalti pimennossa (Shilman, 2016).

Verrattaessa vuoden 2016 tuloksia vuoden 2017 tuloksiin, on tietoisuus muista alustoista kasvanut. Näistä parhaana esimerkkinä on Jest, jonka osalta sekä tietoisuus, että sitä kokeilleiden kehittäjien määrä on selkeässä nousussa. Jest myös, yhdessä Enzymen kanssa, erottuu joukosta korkean tyytyväisyys arvonsa ansiosta (Shilman, 2016).



Kuva 5 JavaScript testing framework satisfaction 2016 (Shilman, 2016)



Kuva 6 JavaScript testing framework satisfaction 2017 (Benitte, Grief & Rambeau, 2017)

Vitali Zaidman antaa myös oman yhteenvetonsa yksikkötestausalustoista artikkelissaan An Overview of JavaScript Testing in 2018.

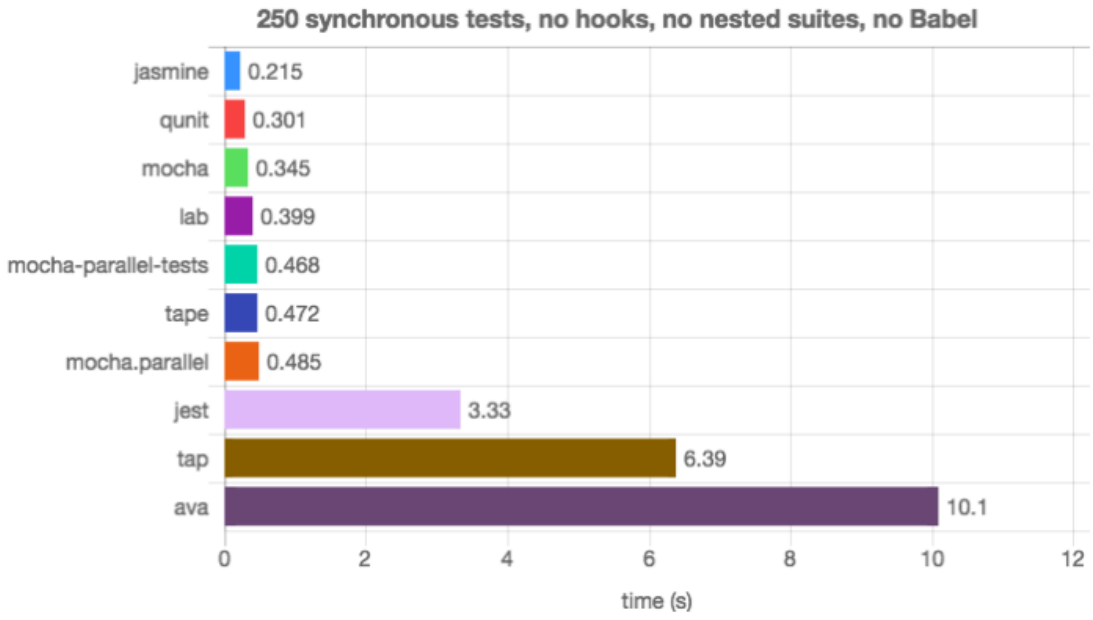
> ** In short, if you want to "just get started" or looking for a fast framework for large projects, go with **Jest**.*
>
> ** If you want a very flexible and extendable configuration, go with **Mocha**.*
>
> ** If you are looking for simplicity go with **Ava**.*
>
> ** If you want to be really low-level, go with **tape**.*

Kuva 7 Yhteenveto JavaScript yksikkötestausalustoista (Zaidman, 2018)

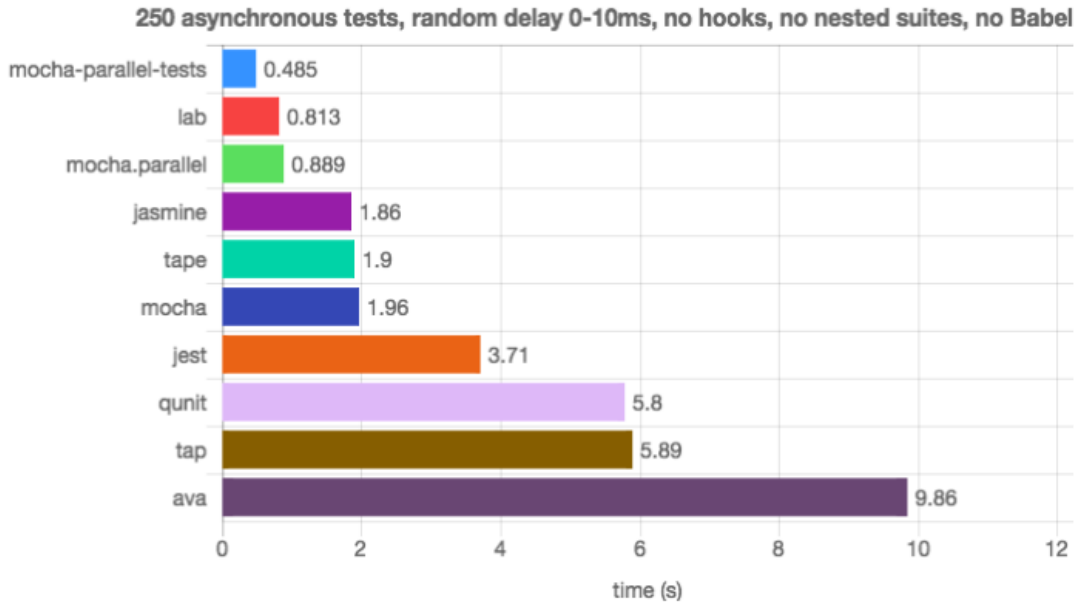
Jest saa korkean suosituksen helpon konfiguraationsa ansiosta. Jest tarjoaa laajat ominaisuudet, mutta on mahdollista lisätä projektiin myös pienellä konfiguraatiolla. Mocha on laajemmin käytetty, mutta vaatii enemmän aikaa projektiin lisäämistä varten. Mocha sisältää vähemmän toiminnallisuuksia itsessään, mutta on laajennettavissa erinäisten lisäpakettien kautta. Ava on kahta edellistä yksinkertaisempi, eikä lisää globaaleja muuttujia. Tape on vieläkin minimaalisempi, ja tarjoaa vain kaiken testaamiseen välttämättömimmän. (Zaidman, 2018) Jest ja Mocha ovat omat valintani yksikkötestaustyökalujen osalta. Jest on omalta osaltaan ominaisuuksiltaan kattavampi, sisältää laajemmat konfigurointimahdollisuudet, ja sen dokumentaatio on sisällöltään parempi. Mocha on kuitenkin myös hyvä vaihtoehto, ja on paremmin tuettuna muiden testaustyökalujen, kuten Karman, Chutzpahn ja Visual Studion NodeJSToolsin kanssa.

3.9.1 Yksikkötestaustalustojen suorituskyky

Suoritusteho on yksi valintakriteereistä, joka voi vaikuttaa käytettävän testausalustan valintaan. Testit olisi hyvä saada suoritettua mahdollisimman nopeasti, jotta virheet voidaan havaita aikaisemmin. (Potapov, 2017)



Kuva 8 Yksikkötestausalustat nopeustesti, synkroninen (Potapov, 2017)



Kuva 9 Yksikkötestausalustat nopeustesti, asynkroninen (Potapov, 2017)

Suosittujen JavaScript yksikkötestausalustojen suoritustesteistä on vedettävissä muutamia johtopäätöksiä:

- Jasmine, Mocha, Tape ja Lab suoriutuvat hyvin synkronisesta testauksesta,
- Asynkronista testausta varten Mochan wrapper-kirjastot mocha-parallel ja mocha-paralle-tests ovat harkinnan arvoisia yhdessä Labin kanssa,
- Jest ja AVA ovat hitaampia testien suorittamisessa, mutta tarjoavat muita ominaisuuksia, jotka voivat olla eduksi testauksessa. (Potapov, 2017)

4 KÄYTTÖLIITTYMÄN TESTAUSTYÖKALUT

Tässä kappaleessa käydään läpi verkkosovellusten käyttöliittymän testaamista sekä JavaScript-työkaluja, joita voidaan hyödyntää selaimen automatisointiin sekä integraatio- ja käyttöliittymätestien kirjoittamiseen.

4.1 Verkkoapplikaatioiden testauksen automatisointi

Yhä useammat sovellukset kirjoitetaan nykyään web-applikaatioina verkkoselaimille. Näiden sovellusten testaamisesta on tullut yhä suurempi osa- alue, ja näiden testien automatisoimisesta on tulossa yhä enemmän ja enemmän vaadittu edellytys. Testauksen automatisoinnista on useita pitkän tähtäimen etuja testausprosessiin. (SeleniumHQ, 2018)

Automatisoitujen testien kirjoittaminen kuitenkin vie aikaa ja resursseja, joten ennen testien kirjoittamista on harkittava, onko se järkevää. Manuaalinen testaus voi olla parempi vaihtoehto, jos aikataulu on kiireinen, eikä minkäänlaista automaatiopohjaa testaukselle ole olemassa. Sovelluksen käyttöliittymään tulevat suuremmat muutokset voivat myös aiheuttaa sen, että automatisoidut testit joudutaan kirjoittamaan uusiksi. (SeleniumHQ, 2018)

4.1.1 W3C Webdriver

Webdriver on sovellusalustasta sekä ohjelmointikielestä riippumaton rajapinta selaimen ohjaamista varten. Webdriver-protokolla määrittää rajapinnan DOM-elementtien havaitsemista ja manipulointia varten. W3C Webdriver-standardi on johdettu Selenium Webdriver-selainautomaatiorungosta. Pääasiassa standardi on tarkoitettu käyttäjän toimintaa simuloivien testien kirjoittamista varten. (W3C Webdriver, 2018)

4.2 Selenium

Selenium on työkalu selaimen ohjaamista varten. Sen pääsääntöinen käyttökohde on verkkoapplikaatioiden automatisointi testausta varten. Käyttötarkoitus ei kuitenkaan ole rajoitettu vain testaamiseen, vaan Seleniumia voidaan hyödyntää myös mm. verkkosivujen hallintatoimintojen automatisoimiseen. Selenium on useiden eri työkalujen joukko, jotka tukevat testien automatisointia omalla tavallaan. (Selenium, 2018 & SeleniumHQ, 2018) Monet JavaScript funktionaaliset testausalustat, esimerkiksi WebdriverIO ja Nightwatch käyttävät Seleniumia selaimen ohjaamista varten.

4.3 Selenium-webdriver

Selenium-webdriver on Seleniumin oma kirjasto selaimen automatisointia varten. Selenium-webdriver tukee seuraavia selaimia sekä käyttöliittymiä, joilla ne toimivat:

- Google Chrome
- Internet Explorer 7, 8, 9, 10, ja 11

- Firefox: viimeisin-, ja edellinen versio sekä viimeisin-, ja edellinen ESR -versio
- Opera
- HtmlUnit
- PhantomJS
- Android (Selendroid: in tai Appium: in avulla)
- IOS (IOS-driverin tai Appiumin avulla) (SeleniumHQ, 2018)

Selenium: ia voidaan käyttää eri ohjelmointikielillä, kuten Java, C#, Python, Ruby, Perl, PHP, ja JavaScript (SeleniumHQ, 2018). Tässä kappaleessa käsitellään JavaScript-versiota, josta käytetään myös nimitystä WebdriverJS.

Selenium-webdriver on ladattavissa npm: n kautta. Lisäksi se tarvitsee erikseen driverin jokaista testattavaa selainta kohti. Selainten driverien sijainti tulee myös lisätä koneen PATH-ympäristömuuttujaan, jotta Selenium-webdriver kykenee löytämään ne. Selenium-standalone-server toimii välityspalvelimena koodin ja selainten driverien välillä. Sen käyttöä ei kuitenkaan suositella, sillä se lisää yhden ylimääräisen vaiheen jokaisen komennon väliin, mikä hidastaa testien suorittamista. (SeleniumHQ, 2018)

4.3.1 Ominaisuudet

Selenium-webdriver on Seleniumin oma työkalu. Koska Selenium-webdriver vain ohjaa selainta, tarvitaan testien kirjoittamista varten erillinen framework. Testikoodi on asynkronista, joten se joudutaan huomioimaan testejä kirjoitettaessa. Testattava selain määritellään myös testikoodissa, joten jokaisessa testissä pitää huomioida kaikki selaimet, joille testit halutaan suorittaa. Selainten asetusten määrittämistä ei kuitenkaan ole dokumentoitu selkeästi, ainakaan JavaScriptin osalta, mikä on turhauttavaa.

Chromen ja Firefoxin käynnistäminen onnistuu automaattisesti, jos selaimet on määritetty koodissa. Edge ja Internet Explorer-selainten käynnistäminen ei onnistunut vastaavalla koodilla. Näiden testaaminen onnistuu kuitenkin ainakin selenium-standalone serveriä käyttäen. Selenium-webdriver: in JavaScript implementaatiot puutteina ovat heikko dokumentaatio sekä selainten konfiguraatio. Konfiguraatio tulee määrittää testin yhteydessä, sen sijaan, että se olisi mahdollista tehdä vaikkapa erillisessä tiedostossa.

4.4 WebdriverIO

WebdriverIO sisältää oman implementaationsa Selenium Webdriverista (Zaidman, 2018). WebdriverIO mahdollistaa selaimen tai mobiilisovelluksen ohjaamisen koodilla. Integroitu testiajuri mahdollistaa asynkronisten komentojen kirjoittamisen synkronisessa muodossa. WebdriverIO myös hoitaa Selenium-serverin hallinnan säästäen käyttäjältä työlään konfiguraation. (WebdriverIO, 2018)

WebdriverIO on kirjoitettu kokonaan JavaScriptillä. Elementtien hakeminen sivulta onnistuu natiiveja JavaScript funktioita käyttäen. Webdriver mahdollistaa myös kuvakaappausten ottamisen sivuista virhetilanteiden sattuessa. WebdriverIO: lla on mahdollista ajaa useita Selenium-sessioita yksittäisessä testissä, mikä mahdollistaa kahden käyttäjän välisten toimintojen, kuten esimerkiksi chat-sovelluksen testaamisen. (WebdriverIO, 2018)

WebdriverIO: n kirjastoa käytetään selaimen ohjeistamiseen, mutta testien suorittamiseen tarvitaan erillinen framework. WebdriverIO tukee Google Chrome, Mozilla Firefox, Opera, ja Safari selaimia. Testit on mahdollista suorittaa samanaikaisesti usealla selaimella.

WebdriverIO: n ominaisuudet

- W3C webdriverin implementaatio, ei ole riippuvainen WebdriverJS: stä.
- Suunniteltu laajennettavaksi ja testausalustoista riippumattomaksi.
    - Vaativat kuitenkin erillisen WebdriverIO-adapterin, joten tällä hetkellä tuettuina ovat Jasmine, Mocha, ja Cucumber (WebdriverIO, 2018).
- CLI, jolla voidaan luoda helposti luoda konfiguraatiotiedosto. Tämä tekee työkalun konfiguroinnista helpompaa
- Webdrivercss web sovelluksen ulkoasun regressiotestausta varten.
- Implementaatio poikkeaa jonkin verran Selenium-webdriverista. (Vijay, 2016)

4.5 Nightwatch

Nightwatch on helppokäyttöinen Node.js käyttöliittymän testaustyökalu selainpohjaisille applikaatioille ja verkkosivuille. Nightwatchin komennon ja assertiot on kehitetty W3C Webdriverin API: n mukaisesti. (Nightwatch, 2018)

Helppo, mutta tehokas syntaksi, joka mahdollistaa testien kirjoittamisen nopeasti käyttäen JavaScript: iä ja CSS-tai Xpath-valitsimia. Testejä on mahdollista suorittaa sarjassa, rinnakkain, yhdessä, ryhmittäin, tunnisteen mukaan tai yksittäin. Nightwatch myös mahdollistaa Selenium-serverin automaattisen ohjaamisen erillisessä aliprosessissa. Tuki pilvitestauspalveluille, kuten SauceLabs, ja BrowserStack. (Nightwatch, 2018)

Toimintojen suorittamista varten ei tarvitse erikseen määrittää selainta odottamaan toimintojen valmistumista, vaan tarvittavat toiminnot voidaan vain linkittää peräkkäin. Jokaiselle selaimen ohjeistamisfunktiolle voidaan myös antaa parametrinä callback-funktio, jossa suorittaa toiminnon suorittamisen jälkeen tarvittava koodi, esimerkiksi haluttu testaus.

Nightwatchin ominaisuuksia:

- Testien suorittaminen useassa eri selaimessa samanaikaisesti. Testeillä voidaan varmistaa, että sovellus toimii kuten pitää halutuilla selaimilla.
- Sisältää oman testausympäristön. Nightwatchin API sisältää selaimen ohjaamiseen tarkoitettujen funktioiden lisäksi funktiot testien kirjoittamiseen.

- Ei tukea ulkoisille testauskirjastoille. Testifunktiot ja -assertiot ovat sidoksissa selainta ohjaaviin metodeihin.
- Helposti laajennettavissa. Mahdollisuus omien komentojen ja assertioiden lisäämiseen.
- Vähäisempi tuki, kuin WebdriverIO: lla. Ei yhtä laajalta käytetty ja siten pienempi käyttäjäyhteisö. (Vijay, 2016)

4.6 Protractor

Protractor on integraatio-/ käyttöliittymän testausframework Angular- ja AngularJS-sovelluksille. Protractor ajaa testejä sovellusta vastaan selaimessa. Protractor on rakennettu WebdriverJS: n (selenium-webdriver) päälle, joka käyttää natiiveja eventejä, ja selainkohtaisia ajureita sovelluksen kanssa toimimiseen. Protractor on lähtökohtaisesti suunniteltu Angular-sovellusten testaamista varten, ja mahdollistaa Angular-komponenttien testaamisen ilman erillistä konfiguraatiota. Protractor osaa odottaa automaattisesti, kunnes verkkosivu on suorittanut kaikki meneillään olevat tehtävät, joten koodissa ei tarvitse erikseen huolehtia testien ja sivun synkronoinnista. (Protractor, 2018)

Protractorille tärkeimpiä ominaisuuksia ovat

- Käyttää Selenium-standalone serveriä, jonka hallintaa kuitenkin on helpotettu mukana tulevalla hallintatyökalulla. Protractorille on mahdollista konfiguroida hallinnoimaan webserverin automaattinen käynnistäminen.
- Tukee Google Chrome, Mozilla Firefox, Internet Explorer sekä Safari-selaimia
- Angular tuki. Protractor on Angular tiimin työkalu integraatio- ja funktionaalista-testausta varten, ja soveltuu parhaiten Angularin testaukseen.
- Protractor tukee Jasmine, Mocha ja Cucumber työkaluja.
- TypeScript tuki. Testejä voidaan kirjoittaa TypeScriptiä käyttäen pienellä konfiguroinnilla.
- Ei tukea mobiilisovellusten testaukselle
- Rakennettu WebdriverJs: n päälle => ylimääräinen kerros Seleniumin ja Protractorin välille, ja on siten riippuvainen WebdriverJs: stä. (Protractor, 2018, Vijay, 2016)

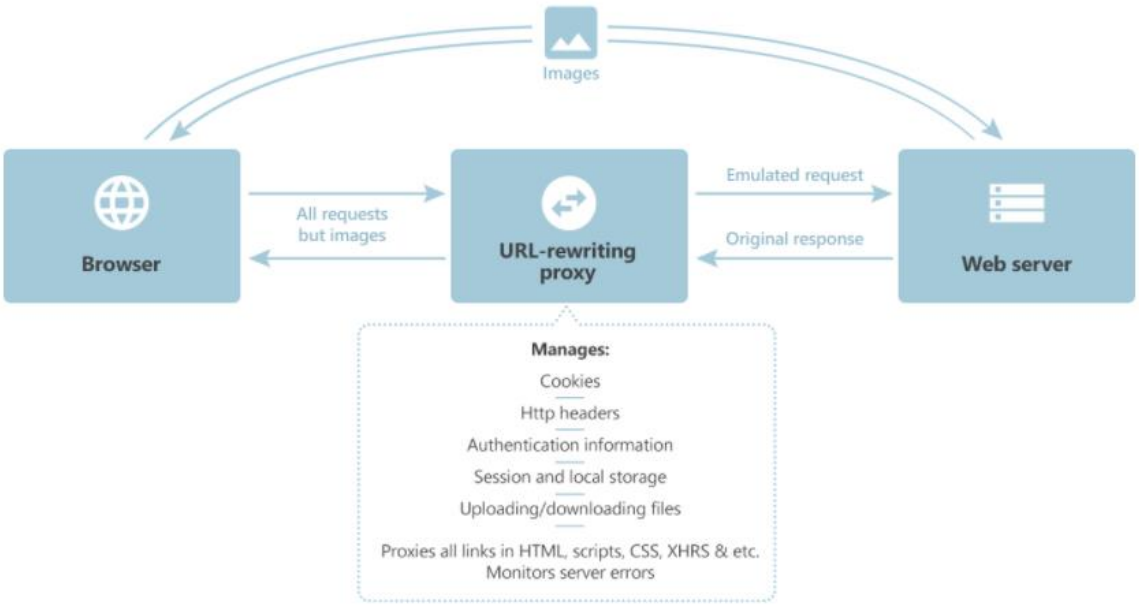
4.7 TestCafe

TestCafe on Node.js työkalu käyttöliittymän testaamisen automatisointiin. Testit voidaan kirjoittaa JavaScriptillä tai TypeScriptillä. TestCafen käyttö ei vaadi erikseen Selenium-webdriveriä. Tämä mahdollistaa, sen, ettei TestCafen toiminnallisuus ole rajattuna webdriverin puutteiden takia. TestCafe tukemat selaimet ovat: Google Chrome, Internet Explorer (9+), Microsoft Edge, Mozilla Firefox, Safari, Android Browser ja Safari mobile. TestCafe tukee testausta myös headless-selaimilla. (TestCafe, 2018)

TestCafesta löytyy sekä maksullinen, että ilmainen open source versio. Maksullisen version ominaisuuksiin kuuluvat mm. hallintapaneeli testien kirjoittamista, muokkaamista, ja ajamista varten. Maksetun version kehittäminen kuitenkin lopetettiin 2015, ja uusi samanniminen open

source versio julkaistiin 2016. Uudempi versio ei sisällä omaa graafista käyttöliittymää, vaan testit voidaan ajaa komentokonsolin kautta. (TestCafe, 2018)

TestCafe käyttää Selenium: in sijasta URL-uudelleenkirjoitus-välityspalvelinta (URL-rewriting-proxy). Välityspalvelin injektoi käyttäjän toimintaa emuloivan scriptin testattavaan sivuun. Tämä mahdollistaa kaiken testaamiseen tarvittavan: käyttäjän toiminnan emuloinnin, todennuksen, scriptien suorittamisen jne. Testattava sivu puolestaan ei koe mitään häiriötä. (Moskovkin, 2017)



Kuva 10 URL- rewriting- proxy (Moskovkin, 2017)

TestCafe sisältää joitain ominaisuuksia, joita olisi vaikeaa implementoida Seleniumin kanssa. Näitä ovat esimerkiksi eristetty testiympäristö, sivun ja html-elementtien latautumisen automaattinen odottaminen ja testien suorittaminen mobiililaitteilla. (Moskovkin, 2017)

Webdriverista riippumattomuus tekee konfiguroinnista yksinkertaisempaa. Html-elementtien etsimistä varten on TestCafella tarjolla oma syntaksi, mikä mahdollistaa elementtien etsimisen css-attribuuttien lisäksi esimerkiksi niiden sisällön mukaan. Käyttäjän toimintojen linkittäminen on myös hyvin yksinkertaista. Ei ole kuitenkaan hyvä, että TestCafe lataa testattavan sivun ennen jokaista testiä. Tätä ominaisuutta ei ole mahdollista poistaa käytöstä. Tämä hidastaa testien suorittamista, sillä sivun uudelleenlataamiseen kuluvan ajan lisäksi voidaan myös joutua toistamaan samoja toimintoja. Vaihtoehtona on toki kirjoittaa useampia testitapahtumia samaan testifunktioon. Useamman testitapauksen kirjoittaminen samaan funktioon ei kuitenkaan ole paras käytäntö. Vaikka tuki testien suorittamiseen mobiililaitteilla on mahdollista, on testien suoritusnopeus kuitenkin valitettavan hidasta.

TestCafen tärkeimpiä ominaisuuksia

- Ei riippuvainen Seleniumsta. Webdriveria ei tarvitse erikseen asentaa testien suorittamista varten.
- Vähäinen konfiguraatio => helppo jo nopea ottaa käyttöön.
- Tuki kaikille selaimille. Mahdollista varmistaa, että vaaditut testit suoriutuvat halutuilla selaimilla.
- Tuki mobiililaitteille. Web-sovelluksen toimintaa voidaan testata myös mobiililaitteilla.
- Remote testaus, jolla voidaan helposti jakaa testi paikallisverkossa muille laitteille.
- Testien rinnakkaissuorittaminen
- Oma testistruktuuri. Testikoodin rakenne eroaa jossain määrin selenium-pohjaisten työkalujen rakenteesta. (Zaidman, 2018)

4.8 Cypress

Cypress on nykyaikaiseen web-kehitykseen rakennettu seuraavan sukupolven Front-End-testausalusta. Cypress yksinkertaistaa testausympäristön asettamisen, testien kirjoittamisen, suorittamisen, ja debuggaamisen. (Cypress, 2018)

Cypress mahdollistaa integraatio- ja funktionaalisen testauksen. Cypress ei käytä testaukseen erikseen Seleniumia. Cypress yhdistää useita eri open source työkaluja, kuten Mocha, Chai ja Sinon, joten testaajan ei tarvitse huolehtia kaikkien tarvittavien osien valitsemista, sillä kaikki tarvittava tulee Cypressin mukana. Cypress tallentaa kuvakaappauksia suoritetuista testeistä ajon aikana, jolloin niitä on mahdollista katsella jälkikäteen. Cypress osaa myös automaattisesti odottaa komentojen ja assertioiden suoriutumisen. (Cypress, 2018)

Cypressin merkittävimpiä ominaisuuksia

- Graafinen käyttöliittymä. Testien ja tulosten hallinnointi ja lukeminen tarkoitukseen luodun käyttöliittymän kautta terminaalin sijasta.
- Ei tarvitse webdriveria. Testejä varten ei tarvita koneelle asentaa erikseen webdriverin instanssia.
- Cypressistä puuttuu vielä monia ominaisuuksia, kuten testien rinnakkaisajo, jotka on tarkoitus lisätä tulevaisuudessa.
- Tuki tällä hetkellä vain Google Chromelle, mutta muiden selainten tukeminen on työn alla.
- Dokumentaatio on selkeä ja kattava.
- Testirakenteena käytetään Mochaa ja assertio-funktioina chai-kirjastoa.
- Testien tulokset samassa ikkunassa testattavan sivun kuvakaappausten kanssa. Tuloksista on helpompaa hahmottaa tarkalleen missä kohtaa sovellusta virhe esiintyy. (Zaidman, 2018 & Cypress 2018)

4.9 Puppeteer

Puppeteer on Google: n kehittämä Node.js kirjasto, joka tarjoaa korkean tason API: n headless Chromen tai Chromium: in ohjaamista varten DevTools-protokollan avulla. Puppeteer kykenee suorittamaan lähes kaiken mitä selainta käyttävä henkilökin, kuten luoda kuvakaappauksia, ja PDF-

versioita verkkosivuista, automatisoida formien lähettämistä, UI: n testaamista, näppäimistön syöttöä, kerätä diagnostiikkatietoja verkkosivun suorituskyvystä, jne. (Puppeteer, 2018)
Googlen ylläpitämä, ja sillä on käytettävissä kaikki viimeisin Chromen tukema toiminnallisuus. Vaikka Puppeteer on vielä uusi työkalu, tulee se hyvällä todennäköisyydellä olemaan ylläpidettynä pitkään.

Puppeteerin ominaisuuksia

- Google Chrome-tiimin ylläpitämä. Kirjastoa ylläpidetään tuettuna viimeisimmän Chromen version kanssa.
- Ei erillistä testiajuria. Testien kirjoittamisen voidaan käyttää samaa työkalua, kuin yksikkötestaamiseen.
- Vain Chrome: n testaaminen. Ei mahdollisuutta testata sovelluksen toimintaa muilla selaimilla.

4.10 Funktionaalisten testausalustojen vertailu

TestCafe, Cypress, ja Puppeteer mahdollistavat funktionaalisen testaamisen ilman erillistä Webdriver-instanssia, mikä tekee niiden projektiin lisäämisestä helppoa. Niillä pystyy aloittamaan testien kirjoittamisen käytännössä lisäämällä vain paketin projektiin, ja konfiguroimalla testitiedostojen sijainnin. Muut tässä raportissa käsitellyt UI: n testaustyökalut hyödyntävät selaimen ohjaamiseen Seleniumia. Webdriverin hallinta ei kuitenkaan ole erityisen vaikeaa muidenkaan testausalustojen kanssa. Osalla testausalustoista, kuten Nightwatch ja Protractor on mahdollista konfiguroida Selenium-webdriverin automaattinen hallinta testien yhteydessä. Tätä varten joudutaan kuitenkin erikseen lataamaan selenium-webdriver sekä haluttavien selainten lisäosat koneelle. Huomioitavaa on myös, että Selenium-standalone-server tarvitsee Javaa (JDK 8+) toimiakseen. Jos testattavassa projektissa hyödynnetään Nodea, on Webdriverin hallintaa mahdollista automatisoida myös erillisen npm-paketin avulla.

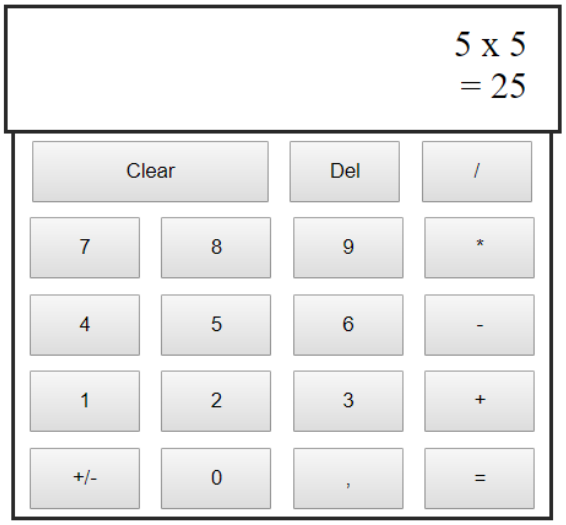
Funktionaalisten testien kirjoittamista varten valittavan testityökalun valinnassa olisi hyvä ottaa huomioon muutamia kriteerejä: onko testattavassa koodissa käytetty Angularia, Reactia, jne, mitä selaimia halutaan testata, ja halutaanko testit kirjoittaa tiettyä kirjastoa käyttäen. (Farhadi, 2018). TestCafe ei tarvitse erillistä konfiguraatiota, ja mahdollistaa testaamisen eri selaimilla. Protractor on kehitetty Angular, ja AngularJS-sovellusten testaamista varten. Puppeteer ei tue kuin yhtä selainta, mutta antaa testaajalle täyden vapauden testauskirjastojen valitsemiseen.

5 ESIMERKKISOVELLUS

Tässä kappaleessa esitellään esimerkkisovelluksen avulla JavaScript-koodin testausta.

5.1 Sovelluksen esittely

Helpoin tapa testata yksikkötestausalustojen toimintaa on kokeilla niiden käyttöä, joten testausalustojen kokeilemista varten, kirjoitettiin yksinkertainen laskin-sovellus. Sovelluksen toteuttamiseen on hyödynnetty TypeScriptiä ja Angularia. Sovelluksen toiminta on hyvin yksinkertainen. Laskimelle syötetään jokin luku, valitaan haluttu laskutoimitus, syötetään toinen luku, ja painetaan =-nappia tuloksen laskemiseksi.



Kuva 11 Laskin-sovellus

Sovellukselle kirjoitettiin yksikkötestit kaikilla työssä käsitellyillä yksikkötestausalustoilla. Myös sovelluksen yksikkötestit on kirjoitettu TypeScriptillä. Kappaleessa käsitellään laskimen logiikan sisältävää calculator-luokkaa, ja sille kirjoitettuja testejä.

```
export class Calculator{
    readonly OPERATOR_SUM = '+';
    readonly OPERATOR_MINUS = '-';
    readonly OPERATOR_MULTIPLY = 'x';
    readonly OPERATOR_DIVISION = '/';
    readonly OPERATOR_EQUALS = '=';
    readonly DECIMAL_SEPARATOR = '.';
    private res: number = undefined;

    public add(num1: number, num2: number){
        this.res = num1 + num2;
        return this.res;
    }
    public subtract(num1: number, num2: number){
        this.res = num1 - num2;
        return this.res;
    }
    public multiply(num1: number, num2: number){
        this.res = num1 * num2;
        return this.res;
    }
    public divide(num1: number, num2: number){
        if(num2 === 0){
            this.res = undefined;
            throw new Error('Division by 0');
        }
        else{
            this.res = num1 / num2;
        }
        return this.res;
    }
    public getLastResult(){
        if(this.res != undefined){
            return new Promise((resolve, reject) => {
                setTimeout(function(res){
                    resolve(res);
                }(this.res), 500);
            });
        }
        else{
            throw new Error('No last result');
        }
    }
    public getHistory(){
        // no implementation
        return null;
    }
}
```

Kuva 12 calculator.ts

5.2 Yksikkötestauksen esittely

Tässä kappaleessa käsitellään yksikkötestien kirjoittamista Jest-testausalustalla sovelluksen Calculator-luokalle. Jest voidaan lisätä Node-projektiin npm install jest -komennolla. Jest lisää oman testiympäristönsä sekä assertionfunktionsa projektiin globaaleina muuttujina, joten testien kirjoittamista varten tarvitaan testitiedostoon lisätä vain testattava moduuli. Lisääminen tiedostoon onnistuu Noden import tai require lauseilla. Huomioitavaa on myös, että laskin luokka on määritelty export-lauseella, mikä mahdollistaa sen lisäämisen muihin tiedostoihin.

```
import { Calculator } from '../../../app/calculator';

describe('Calculator class', function() {

    var calculator: Calculator;

    beforeEach(function(){
        calculator = new Calculator();
    });
```

Kuva 13 Testin alkumääritykset

Calculator luokalle on testissä luotu muuttuja, johon määritellään uusi olio ennen jokaista testiä beforeEach-funktiolla. Tällä varmistetaan, että laskimen tila on sama jokaisessa testissä. Ensimmäisenä testissä varmistetaan, että Calculator-luokka on olemassa. expect(calculator).toBeDefined()-metodi tarkistaa, että sille annettu parametri on määritelty.

```
test('should be defined', function() {
    expect(calculator).toBeDefined();
});
```

Kuva 14 Testataan luokan olemassaolo

Luokalla on funktiot add, subtract, multiply, ja divide peruslaskutoimitusten suorittamista varten. Jokaiselle laskutoimitukselle on kaksi erillistä testiä. Näillä varmistetaan, että laskin palauttaa oikeat tulokset sekä positiivisilla, että negatiivisilla luvuilla. Kaikki kuvan 15 assertiot voitaisiin määrittää yksittäisessä testifunktiossa. Yksittäisessä testifunktiossa olisi kuitenkin hyvä käyttää mahdollisimmän vähän assertio-funktiota. Jest, kuten jotkut muutkin testausalustat, ei erikseen ilmoita jokaista virheellistä assertiota, vaan virheellisen testifunktion. Tästä johtuen, jos testi sisältää useamman virheellisen assertion, raportoi Jest vain ensimmäisestä virheellisestä assertiosta.

```
describe('should have an add function', function(){
    test('should add postestive numbers', function() {
        expect(calculator.add(1, 1)).toBe(2);
        expect(calculator.add(0, 2)).toBe(2);
    });
    test('should add negative numbers', function() {
        expect(calculator.add(2, -5)).toBe(-3);
        expect(calculator.add(-3, -5)).toBe(-8);
    });
});
```

Kuva 15 Lukujen yhteenlaskun testaaminen

expect(actual).toBe(expected) vertaa sille syötettyjä arvoja loogisella operaattorilla ===, ja palauttaa true, jos annetut arvot vastaavat toisiaan. Näillä testeillä varmistetaan, että laskin palauttaa oikein lasketut arvot, kun sille syötetään oikeita arvoja. Oleellista on myös testata, toiminta virhetilanteissa. Laskimella on mahdollista syöttää lukuja vain numeropainikkeita painamalla, joten ei ole tarpeellista huolehtia siitä ovatko syötetyt arvot tyypiltään lukuja. Testattava virhetilanne on kuitenkin 0: lla jakaminen. Yritettäessä jakaa lukua 0: lla laskimen ei tulisi suorittaa laskua vaan nostaa siitä virhe.

```
test('should thrown an error when trying to divide by zero', function(){
    expect(function() {
        calculator.divide(2, 0);
    }).toThrowError('Division by 0');
});
```

Kuva 16 0:lla jakamisen testaaminen

Ylimääräisenä funktiona luokalla on vielä getLastResult, joka palauttaa edellisen lasketun arvon 0,5 sekunnin viiveellä. Kyseistä funktiota ei käytetä laskin-sovelluksessa, vaan se on olemassa vain asynkronisen testin esittämistä varten.

```
test('should return the previous result', function() {
    calculator.add(2, 2);
    calculator.getLastResult().then(function(res){
        expect(res).toBe(4);
    });
});
```

Kuva 17 getLastResult-funktion testaus

getLastResult-metodi palauttaa JavaScript Promise()-olion, joten palautetun olion arvon oikeellisuus voidaan testata .then()-metodissa. Jest mahdollistaa Promise-syntaksin käsittelyn ilman erillistä määritystä.

Viimeinen testi demonstroi mockin käyttöä funktion korvaamiseen testaamista varten. Oletetaan, että suoritetut laskutoimitukset halutaan tallentaa, jotta niitä voidaan myöhemmin tarkastella uudestaan. getHistory()-funktio voisi esimerkiksi hakea ennestään lasketut arvot tietokannasta. Tätä toiminnallisuutta ei kuitenkaan ole implementoitu sovellukseen, vaan getHistory() palauttaa null-arvon.

```
public getHistory(){
    // no implementation
    return null;
}
```

Kuva 18 getHistory()-funktio

Yksikkötestissä halutaan testata vain yksittäistä koodin osaa, joten ulkoisia riippuvuuksia, kuten http kutsuja tai tietokannan käsittelyä ei haluta käyttää testauksessa. Test double-funktioita voidaan käyttää korvaamaan jokin ongelmia aiheuttava koodin osa. Jest sisältää spy-, ja test double-funktioita, joita voidaan käyttää tähän tarkoitukseen.

```
test('getHistory() a mock function to return previous calculation values', function() {
    spyOn(calculator, 'getHistory').and.returnValue(['10', '0', '25', '1']);
    calculator.add(5, 5);
    calculator.subtract(5, 5);
    calculator.multiply(5, 5);
    calculator.divide(5, 5);
    expect(calculator.getHistory()).toEqual(['10', '0', '25', '1']);
});
```

Kuva 19 Jest spy-demonstraatio

spyOn-metodilla voidaan korvata haluttu funktio. Kuvan 19 toteutuksessa getHistory korvataan test doublella, ja määritetään se palauttamaan kutsuttaessa määritetty arvo. Test double korvaa funktion todellisen toteutuksen, joten kuvan 19 testi menee läpi, vaikka getHistory-funktio ei todellisuudessa palauttaisikaan oikeaa arvoa.

5.3 Yksikkötestien suorittaminen

Jest testit voidaan suorittaa komentokonsolin kautta, suorittamalla haluttu testitiedosto Jest node paketilla. Jest on oltava asennettuna lokaalina pakettina projektiin. On myös mahdollista asentaa Jest globaalina npm-pakettina, jolloin Jest: in CLI-komennot ovat käytettävissä. TypeScriptillä kirjoitettujen testien suorittamista varten tarvitaan myös erillinen ts-jest-moduuli. Ts-jest mahdollistaa TypeScriptillä kirjoitettujen testien suorittamisen Nodella ilman että koodi käännettäisiin ensin erikseen JavaScriptiksi.

```
"jest": {
  "transform": {
    "^.+\\.tsx?$": "ts-jest"
  },
  "testMatch": [
    "**/?(*.)(spec).(js|ts)"
  ],
  "testPathIgnorePatterns": [
    "/build//"
  ],
  "moduleFileExtensions": [
    "ts",
    "js",
    "json"
  ]
},
```

Kuva 20 Jest konfiguraatio

Kuvan 20 konfiguraatiossa on määritettynä testMatch-asetuksella jest etsimään spec.ts/js- päätteisiä tiedostoja. Lisäksi npm-komennolla, jolla testit suoritetaan, on annettu testPathPattern-asetuksella polku testitiedostojen sijaintiin. Koko polku testitiedostoinen voitaisiin myös määrittää Jest: in konfiguraatiossa. Kuvassa 21 on package.json-tiedostossa määritetty komento testien suorittamista varten.

```
"jest": "jest --testPathPattern=src/test/unit/jest/",
"jest-c": "jest --coverage --testPathPattern=src/test/unit/jest/",
```

Kuva 21 npm-skripti testien suorittamista varten

Testit voidaan nyt ajaa suorittamalla projektin kansiossa komentokonsolin kautta komento npm run jest. Jest-c-scriptissä on lisäksi määritelty coverage-asetus, joka raportoi myös yksikkötestien kattavuuden testien suorittamisen yhteydessä.

```
 PASS  src\test\unit\jest\calculator.jest.spec.ts
 PASS  src\test\unit\jest\calculator.component.jest.spec.ts

Test Suites: 2 passed, 2 total
Tests:       28 passed, 28 total
Snapshots:   0 total
Time:        3.482s, estimated 6s
Ran all test suites matching /src\\test\\unit\\jest\\/i.
```

Kuva 22 Raportti läpi menneistä testeistä

5.4 Käyttöliittymän testauksen esittely

Tässä kappaleessa käsitellään esimerkkisovelluksen funktionaalista testaamista. Toisin kuin yksikkötestauksessa, käyttöliittymää testattaessa ei ladata yksittäisiä tiedostoja, vaan sovellus laitetaan käyntiin, ja käyttöliittymän testaustyökalulla ajetaan scripti, jolla selain suorittaa automaattisesti sille määritetyt toiminnot. Kuten yksikkötestauksessa, selaimen suoritettua halutut toiminnot voidaan tarkastella, että tietyillä muuttujilla, tai kentillä on halutut arvot, selain on navigoinut oikeaan url-osoitteeseen, jne.

Esimerkkisovelluksen testaamista käsitellään Nightwatch-testausalustalla. Kuten Jest, myös Nightwatch on saatavilla npm: n kautta, joten se voidaan ladata Node-projektiin npm install nightwatch- komennolla.

```
module.exports = {

    before: function(browser) {
        browser
            .url('http://localhost:4200/');
        },

    beforeEach: function(browser) {
        browser
            .click(b_clear);
        },
    after: function(browser) {
        browser
            .closeWindow()
            .end();
        },
```

Kuva 23 Nightwatch test-hooks

Nightwatchilla on mahdollistaa määrittää before, beforeEach, after, ja afterEach-funktiot haluttujen toimintojen suorittamista varten ennen testejä tai niiden jälkeen. Kuvassa 23 on hyödynnetty before-metodia testattavan sivun url-osoitteeseen navigoimiseen. beforeEach-funktiossa painetaan laskimen clear-näppäintä tilan nollaamista varten. after-funktio suoritetaan kaikkien testien jälkeen, joten siinä suljetaan selaimen ikkuna, ja lopetetaan sessio.

```
'Calculator title' : function (browser) {
    browser.assert.containsText('#title', 'Calculator')
},

'should update  display when numeric values are pressed' : function (browser) {
    browser
        .click(b_1, function(browser){
            this.expect.element(display_res).text.to.contain('1');
        })
        .click(b_2, function(browser){
            this.expect.element(display_res).text.to.contain('12');
        });
},
```

Kuva 24 Nightwatch otsikon ja lukujen syöttämisen testaus

Kuvassa 24 on kaksi Nightwatchilla kirjoitettua testifunktiota. Testifunktioiden rakenne on hyvin tuttu yksikkötestien vastaavista funktioista. Testille annetaan ensimmäisen parametrinä kuvaus, ja toisena funktio. Funktiolle annetaan parametrinä browser-objekti, jota käytetään selaimen ohjaamista varten. Ensimmäisessä testissä varmistetaan, että applikaation otsikko elementti sisältää oikean tekstin. Otsikko voidaan valita sivulta css-id-arvon avulla. b_1 ja b_2, ja display-res-

muuttujat sisältävät sivun html-elementtien id-arvot. click metodilla ohjeistetaan selainta klikkaamaan määritettyä elementtiä. Click-tapahtumalle voidaan antaa toisena parametrinä callback-funktio, jos on tarvetta tehdä jotain painikkeen painamisen jälkeen. Toisessa testissä testataan, että applikaatio lataa oikean arvon näytölle. Selaimelle voidaan antaa myös useita komentoja peräkkäin, joiden jälkeen suoritetaan useampi assertio, kuten kuvassa 25.

```
'clicking = should show the entered calculation and the calculated value on the display': function(browser){
    browser
        .click(b_6)
        .click(b_0)
        .click(b_minus)
        .click(b_8)
        .click(b_equal, function(browser){
            this.expect.element(display_calc).text.to.contain( "60 - 8");
            this.expect.element(display_res).text.to.contain("= 52");
        })
},
```

Kuva 25 Laskutoimituksen testaus

5.5 Funktionaalisten testien suorittaminen

Nightwatchin testit voidaan käynnistää komentokonsolin kautta. Kuten Jestille, on myös määrittänyt Nightwatchille omat npm scriptit testien suorittamista varten. Nightwatchille määritetään konfiguraatio nightwatch.json-tiedostossa, joka annetaan –config avaimella parametrina käynnistettäessä Nightwatch konsolilla. Komennossa voidaan myös määrittää selain, jolla testit suoritetaan.

```
"nightwatch": "nightwatch --config ./nightwatch.json -e chrome",
"nightwatch-ff": "nightwatch --config ./nightwatch.json -e firefox",
"nightwatch-edge": "nightwatch --config ./nightwatch.json -e edge",
"nightwatch-all": "nightwatch --config ./nightwatch.json -e chrome,firefox,edge",
```

Kuva 26 Nightwatch npm-scripti

Nightwatchilla suoritettavat testitiedostot määritetään nightwatch.json-konfiguraatiotiedostossa. Testitiedostojen lisäksi, on määritettävä myös selain, jolla testit suoritetaan. Nightwatch käyttää selaimen ohjaamista varten selenium-webdriveria. Webdriverin käyttö on mahdollista hoitaa joko manuaalisesti, tai automaattisesti Nightwatchn avulla. Jos halutaan, että Nightwatch hoitaa seleniumin käynnistämisen automaattisesti, tulee konfiguraatiotiedostossa määrittää selenium: start_process-kentälle arvoksi true. Lisäksi tulee määrittää polku sekä selenium-standalone-

serverille, että kaikkien haluttujen selainten liitännäisille. On myös mahdollista määrittää selainten ajureiden sijainti järjestelmän PATH-ympäristömuuttujaan.

```
{
  "src_folders" : ["src/test/functional/nightwatch"],
  "output_folder" : "src/test/functional/nightwatch/reports",
  "custom_commands_path" : "",
  "custom_assertions_path" : "",
  "page_objects_path" : "",
  "globals_path" : "",

  "selenium" : {
    "start_process" : true,
    "start_session" : true,
    "server_path" : "./webdriver/selenium-server-standalone.jar",
    "log_path" : "./webdriver/log/",
    "port" : 4444,
    "cli_args" : {
      "webdriver.chrome.driver" : "./webdriver/chromedriver.exe",
      "webdriver.gecko.driver" : "./webdriver/geckodriver.exe",
      "webdriver.edge.driver" : "./webdriver/MicrosoftWebDriver.exe",
      "webdriver.ie.driver" : "./webdriver/IEDriverServer.exe"
    }
  },

  "test_settings" : {
    "default" : {
      "launch_url" : "http://localhost/4200",
      "selenium_port"  : 4444,
      "selenium_host"  : "localhost",
      "skip_testcases_on_fail": false,
      "end_session_on_fail": false,

      "silent": true,
      "screenshots" : {
        "enabled" : true,
        "path" : "src/test/functional/nightwatch/errorShots"
      }
    },
```

Kuva 27 Nightwatch konfiguraatio 1/2

```
    "chrome" : {
      "desiredCapabilities": {
        "browserName": "chrome",
        "cssSelectorsEnabled": true,
        "chromeOptions": {
          "args": [
            "headless"
          ]
        }
      }
    },
    "firefox" : {
      "desiredCapabilities": {
        "browserName": "firefox",
        "marionette": true,
        "cssSelectorsEnabled": true
      }
    },
    "edge" : {
      "desiredCapabilities": {
        "platform": "Windows 10",
        "browserName": "MicrosoftEdge",
        "version": "16.16299",
        "cssSelectorsEnabled": true
      }
    },
    "ie" : {
      "desiredCapabilities": {
        "browserName": "IE",
        "cssSelectorsEnabled": true
      }
    }

  }
}
```

Kuva 28 Nightwatch konfiguraatio 2/2

```
Starting selenium server... started - PID:  6080

[Calculator Component Nightwatch Spec] Test Suite
=================================================

Running:  Calculator title
 √ Testing if element <#title> contains text: "Calculator".

OK. 1 assertions passed. (3.778s)

Running:  should update  display when numeric values are pressed
 √ Expected element <#displayRes> text to contain: "1"
 √ Expected element <#displayRes> text to contain: "12"

OK. 2 assertions passed. (208ms)

Running:  clicking + should set + as the calculation operator
 √ Expected element <#displayCalc> text to contain: "+"

OK. 1 assertions passed. (144ms)

Running:  clicking - should set - as the calculation operator
 √ Expected element <#displayCalc> text to contain: "-"

OK. 1 assertions passed. (117ms)

Running:  clicking * should set * as the calculation operator
 √ Expected element <#displayCalc> text to contain: "x"

OK. 1 assertions passed. (132ms)

Running:  clicking / should set / as the calculation operator
 √ Expected element <#displayCalc> text to contain: "/"

OK. 1 assertions passed. (114ms)

Running:  clicking = should show the entered calculation and the calculated value on the display
 √ Expected element <#displayCalc> text to contain: "60 - 8"
 √ Expected element <#displayRes> text to contain: "= 52"

OK. 2 assertions passed. (370ms)

Running:  clicking the +/- button should change the current values sign
 √ Expected element <#displayRes> text to contain: "-"
 √ Expected element <#displayRes> text to not equal: "-"

OK. 2 assertions passed. (229ms)

Running:  clicking the +/- button should do nothing if the value is 0
 √ Expected element <#displayRes> text to not equal: "-"

OK. 1 assertions passed. (93ms)

Running:  decimal point should be appended after the current value
 √ Expected element <#displayRes> text to equal: "4."
 √ Expected element <#displayRes> text to equal: "4.2"

OK. 2 assertions passed. (232ms)

Running:  decimal point should not be added if the current value is already a decimal number
 √ Expected element <#displayRes> text to equal: "7.7"

OK. 1 assertions passed. (217ms)

Running:  should delete the last added value, and set the value to 0 when last value is deleted
 √ Expected element <#displayRes> text to equal: "2."
 √ Expected element <#displayRes> text to equal: "0"

OK. 2 assertions passed. (302ms)

Running:  clicking clear should clear any calculations and set res to 0
 √ Expected element <#displayCalc> text to equal: ""
 √ Expected element <#displayRes> text to equal: "0"

OK. 2 assertions passed. (253ms)

OK. 19  total assertions passed. (9.129s)
```

Kuva 29 Nightwatch raportti suoritetuista testeistä

6 LOPPUTULOKSET JA YHTEENVETO

Tässä kappaleessa käydään lyhyesti läpi, mitä työssä opittiin.
Työn tarkoituksena oli tutustua erilaisiin olemassa oleviin JavaScript-koodin testausalustoihin ja -työkaluihin. Kokeilinkin työssä sekä yksikkö-, että käyttöliittymätestien kirjoittamista usealla eri työkalulla. Työn aikana tutustuttiin myös koodin testaamisen etuihin, ja testivetoisen kehityksen malliin. Työ toi selvästi esille miksi testaamisen automatisointia olisi hyvä tehdä, ja mitä etuja siitä olisi saatavissa.

6.1 Testaamisen automatisointi

Automatisoitujen testien perusrakenne on yksinkertainen. Testissä suoritetaan haluttu koodin ja verrataan suoritetun koodin tuloksia odotettuihin arvoihin. Yksikkötestauksessa on kuitenkin jossain määrin merkitystä sillä, kuinka logiikka on koodissa toteutettu. Testien lisääminen koodiin jälkikäteen voi osoittautua haastavaksi. Kun testattava koodi suorittaa selkeästi jonkin yksittäisen toiminnon, on testien kirjoittaminen sille helppoa. Todellisuudessa koodi ei kuitenkaan tule aina olemaan näin yksiselkoista.

Yksikkötestien kirjoittamisessa voi helposti tulla esille ongelmia, jos testattavuutta ei ole huomioitu koodissa. Koodia ei ole jaettu selkeisiin osa-alueisiin vaan yksittäiset funktiot voivat suorittaa useita eri toimintoja. Jos koodi on monimutkaista, joudutaan testeistäkin tekemään monimutkaisempia, mikä voi helposti johtaa virheisiin testikoodissa. Yksikkötesteistä saadaan paras hyöty irti, jos ne kirjoitetaan TDD-mallin mukaisesti ennen varsinaista koodia, tai ainakin samanaikaisesti koodin kanssa. Huomioimalla testattavuus alusta alkaen, saadaan koodi kirjoitettua selkeämpään muotoon.

Käyttöliittymän testaaminen ei riipu mitenkään testattavan koodin logiikasta, mikä tekee käyttöliittymätestien kirjoittamisesta helpompaa. Käyttöliittymätestien ongelmana on kuitenkin niiden riippuvuus testattavan sivun rakenteesta. Käyttöliittymää testaava koodi on riippuvainen mm. sivulla olevien elementtien tunnistamisesta. Selainta ohjaavan koodin tulee kyetä luotettavasti löytämään testattavalta sivulta kaikki tarvittavat elementit, joten pienetkin muutokset käyttöliittymään voivat rikkoa testit. Testien ylläpitäminen voikin tämän takia koitua työlääksi.

Käyttöliittymätestien ylläpidon helpottamiseksi olisikin suositeltavaa ylläpitää esimerkiksi erillistä PageObject-mallia testattavan sivun oleellisista elementeistä. Elementeille voi myös lisätä testaamista varten oman css-tunnisteensa, jota käytetään vain elementin löytämiseen testikoodissa. UI: n testausta varten joudutaan myös ohjaamaan varsinaista selainta, mikä tekee niiden suorittamisesta hidasta. Virheet olisi hyvä havaita mahdollisimman laajalti yksikkötesteissä, jotta testaaminen olisi tehokasta ajankäytön osalta.

6.2 Testaustyökalut

Työssä käytetty esimerkkisovellus on erittäin yksinkertainen, joten sillä on helppoa demonstroida yksikkötestien perusrakenne. Sovelluksen yksinkertaisuuden takia, testien kirjoittamisen haasteet eivät kuitenkaan tule parhaalla mahdollisella tavalla esille. Lisättäessä yksikkötestejä koodin jälkikäteen, voidaan koodia joutua muokkaamaan huomattavasti, jotta koodille saadaan luotua hyödyllisiä testejä.

Työssä kokeilluista työkaluista oma suositus menee Jestille. Jest on ominaisuuksiltaan kattava sekä laajalti konfiguroitavissa. Jestin dokumentaatio on myös erinomainen, ja itse framework on edelleen aktiivisen kehityksen alla. Jest myös tukee Babelia, Typescriptiä sekä eri JavaScript sovelluskehyksiä. Nämä tosin vaativat omat lisämoduulinsa, eivätkä kaikki osa-alueet välttämättä ole tuettuina.

Testien kirjoittaminen muillakin alustoilla on yksinkertaista. Nämä eivät kuitenkaan ole yhtä kattavia toiminnallisuuksiltaan, ainakaan lisäämättä näitä erikseen. Toinen mainittava työkalu on Karma, jolla testit voidaan ajaa suoraan useassa eri selaimessa. Useilla työkaluilla testit ajetaan Node-ympäristössä, mikä ei vastaa koodin suorittamista selaimessa. Testit voidaan ladata selaimessa myös ilman sitä, mutta Karma tekee testien ajamisen selaimessa helpommaksi automatisoimalla prosessia, ja helpottaen testien ja testattavien tiedostojen hallinnointia.

Oleellisempaa kuitenkin olisi automatisoida testejä ainakin jollain alustalla. Uuden testialustan opettelu on huomattavasti helpompaa, jos ennestään osaa käyttää jotain toista. Testien ylläpitäminen myös auttaa parantamaan koodin laatua.

Käyttöliittymän testaamista varten ei työkalun valinta ole yhtä yksiselkoista. Seleniumista riippumattomien työkalujen käyttöön ottaminen on yksinkertaisempaa. TestCafe vaatii minimaalisen konfiguraation, ja sisältää useita eri ominaisuuksia, kuten testien rinnakkaissuorittaminen useassa selaimen instanssissa, jotka tekevät siitä houkuttelevan vaihtoehdon. Testit voidaan myös suorittaa lähes millä tahansa selaimella. Ainoana edellytyksenä on, että testattava selain on myös asennettuna testejä ajavalle koneelle. Toisin kuin seleniumia käyttävillä työkaluilla, ei eri selainten testaamista varten ole tarvetta ladata erikseen ajureita, tai määrittää erillistä konfiguraatiota erikseen jokaisen selaimen testaamista varten. Testattavat selaimet voidaan yksinkertaisesti antaa parametreinä testien käynnistämisen yhteydessä. Jos eri selainten testaamisen mahdollisuus ei ole valintakriteeri, Puppeteer on hyvä vaihtoehto. Puppeteerin kirjastoa voidaan käyttää selaimen ohjaamiseen, ja itse testit voidaan kirjoittaa samaa työkalua käyttäen, kuin yksikkötestitkin. CypressIO on omalta osaltaan myös toimiva vaihtoehto.

Selenium-pohjaisissa työkaluissa joudutaan käyttämään enemmän aikaa konfiguroimiseen. Nightwatch ja WebdriverIO ovat näistä hyvin soveltuvia ratkaisuja. Itse suosin enemmän Nightwatchia, sillä se mahdollistaa selenium-webdriverin automaattisen hallinnan testien käynnistämisen yhteydessä. WebdriverIO: lla ei ole mahdollista konfiguroida samaa, vaan webdriver instanssi täytyy erikseen käynnistää ja sammuttaa. WebdriverIO: lla on kuitenkin laajempi käyttäjäyhteisö. WebdriverIO: lle on tarjolla enemmän lisäosia sen toiminnan laajentamiseksi, kuten esimerkiksi grunt-ja gulp-lisäosat sekä webdrivercss sivun ulkoasun regressiotestausta varten.

Seleniumista riippumattomuus on ehdottomasti etu, mikä tekee käyttöliittymän testausalustasta suositeltavamman vaihtoehdon.

7 LÄHTEET

SearchSoftwareQuality, [Viitattu 2019-05-27] Luettavissa: https://searchsoftwarequality.techtarget.com/

Cary, D. 2010 What Are Assertions, [Viitattu 2019-05-31] Luettavissa: http://wiki.c2.com/?WhatAreAssertions

Paralogarajah, P. What is Mocking in Testing?, [Viitattu 2019-05-31] Luettavissa: https://medium.com/@piraveenaparalogarajah/what-is-mocking-in-testing-d4b0f2dbe20a

MDN Web Docs, [Viitattu 2018-03-01] Luettavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript

Powel, T, A. Schneider, F. 2012 JavaScript The Complete Reference, Third Edition, New York: McGraw-Hill 2012

W3Schools. JavaScript Versions. [Viitattu 2018-03-01] Luettavissa https://www.w3schools.com/js/js_versions.asp

ECMAScript compatibility table. [Viitattu 2018-03-06] Luettavissa http://kangax.github.io/compat-table/

ECMA-262 Standard. [Viitattu 2018-01-03] Luettavissa https://www.ecma-international.org/publications/standards/Ecma-262.htm

Jang, P. 2017 Modern JavaScript Explained For Dinosaurs [Viitattu 2018-03-06] Luettavissa: https://medium.com/the-node-js-collection/modern-javascript-explained-for-dinosaurs-f695e9747b70

Node.js, [Viitattu 2018-03-01] Luettavissa: https://nodejs.org/en/

MDN Web Docs, Express/Node Introduction [Viitattu 2018-03-07] https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Npm, Luettavissa https://docs.npmjs.com/getting-started/what-is-npm

TypeScript, Luettavissa https://github.com/Microsoft/TypeScript

Zaidman, V. 2018. An Overview of JavaScriptTesting in 2018. [Viitattu 2018-02-05]. Luettavissa: https://medium.com/welldone-software/an-overview-of-javascript-testing-in-2018-f68950900bc3

Jorgensen, P. 2016. Software testing: a craftsman's approach, 4th Edition. Auerbach Publishing

Eugene, Y. E. 2010. JavaScript Testing Beginner's Guide: Test and Debug JavaScript the Easy Way. Packt Publishing

Elliot, E. 2016 The Outrageous Cost of Skipping TDD & Code Reviews [Viitattu 2018-02-19]. https://medium.com/javascript-scene/the-outrageous-cost-of-skipping-tdd-code-reviews-57887064c412

Elliot, E. 2016 5 Questions Every Unit Test Must Answer [Viitattu 2018-02-15]. https://medium.com/javascript-scene/what-every-unit-test-needs-f6cd34d9836d

Tang, D. 2016 5 Ways Unit Testing Can Help You [Viitattu 2018-03-07] Luettavissa https://thejsguy.com/2016/01/28/5-ways-unit-testing-can-help-you.html

Elliot, E. 2016. JavaScript Testing: Unit vs Functional vs Integration Tests. [Viitattu 2018-02-19]. Luettavissa https://www.sitepoint.com/javascript-testing-unit-functional-integration/

Cohn, M. 2009, The Forgotten layer of the Test Automation Pyramid. [Viitattu 2018-02-19]. http://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

Hazem, S. 2013, JavaScript unit testing. Packt Publishing

Bushev, Y. 2017.11.14, Top 15 UI Test Automation Best Practices You Should Follow. [Viitattu 2018-03-27]. Luettavissa https://www.blazemeter.com/blog/top-15-ui-test-automation-best-practices-you-should-follow

Sonmez, J. 2010 Back to Basics: Why Unit Testing is Hard. [Viitattu 2018-04-09] Luettavissa https://simpleprogrammer.com/back-to-basics-why-unit-testing-is-hard/

Murphey, R. 2013. Writing Testable JavaScript. Luettavissa http://alistapart.com/article/writing-testable-javascript

Zaeffer, J. 2012 Introduction to JavaScript Unit Testing. Luettavissa https://www.smashingmagazine.com/2012/06/introduction-to-javascript-unit-testing/

Walton, P. 2016. Learning How to Set Up Automated, Cross-browser JavaScript Unit Testing. [Viitattu 2018-04-09] Luettavissa https://philipwalton.com/articles/learning-how-to-set-up-automated-cross-browser-javascript-unit-testing/

Rocheleau, J. 2016. JavaScript Unit Testing For Beginners. Luettavissa https://designmodo.com/test-javascript-unit/

Jasmine. Luettavissa https://jasmine.github.io/

Mocha. Luettavissa https://mochajs.org/

Jest. Luettavissa https://facebook.github.io/jest/

Ava. Luettavissa https://github.com/avajs/ava

Tape. Luettavissa https://github.com/substack/tape

Elliot, E. 2015, Why I Use Tape Instead of Mocha & So Should You. Luettavissa https://medium.com/javascript-scene/why-i-use-tape-instead-of-mocha-so-should-you-6aa105d8eaf4

Tap. Luettavissa http://www.node-tap.org/

Lab. Luettavissa https://github.com/hapijs/lab

QUnit. Luettavissa http://qunitjs.com/

Karma-runner. Luettavissa https://karma-runner.github.io/2.0/index.html

Benitte, R. Greif, S. Rambeau, M. 2017. The State of JavaScript 2017. Luettavissa https://stateofjs.com/

Shilman, M. 2016. The State of JavaScript 2016-Testing Frameworks. Luettavissa http://2016.stateofjs.com/2016/testing/

Potapov, V. 2017, JavaScript Test-Runners Benchmark. Luettavissa https://medium.com/dailyjs/javascript-test-runners-benchmark-3a78d4117b4

SeleniumHQ Luettavissa: https://www.seleniumhq.org/

Selenium, Luettavissa https://w3c.github.io/webdriver/webdriver-spec.html

W3C Webdriver, Luettavissa https://w3c.github.io/webdriver/webdriver-spec.html

Selenium-webdriver, Luettavissa https://github.com/SeleniumHQ/selenium/tree/master/javascript/node/selenium-webdriver

WebdriverIO, Luettavissa http://webdriver.io/

Lamping, K. 2016, 4 Reasons why you should consider using WebdriverIO for testing. Luettavissa: http://blog.kevinlamping.com/4-reasons-why-you-should-think-about-using-webdriverio/

Nightwatch, Luettavissa http://nightwatchjs.org/

TestCafe, Luettavissa https://github.com/DevExpress/testcafe

Moskovkin, A. 2017. TestCafe: An e2e Testing Tool That Doesn’t use Selenium. Luettavissa https://dzone.com/articles/testcafe-e2e-testing-tool

TestCafe Live, Luettavissa https://github.com/DevExpress/testcafe-live

Cypress, Luettavissa https://www.cypress.io/

Chan, E. 2018. Cypress: The Future of end-to-end testing for web applications. Luettavissa https://medium.com/tech-quizlet/cypress-the-future-of-end-to-end-testing-for-web-applications-8ee108c5b255

Protractor, Luettavissa https://www.protractortest.org/#/

Puppeteer, Luettavissa https://github.com/GoogleChrome/puppeteer

Lewis, A. 2017. Top 5 Most Rated Node.js Frameworks for End-to-End Web Testing. Luettavissa https://medium.com/@adrian_lewis/top-5-most-rated-node-js-frameworks-for-end-to-end-web-testing-f8ebca4e5d44

Vijay. 2016. Protractor vs WebdriverIO vs Nightwatch. Luettavissa http://www.webdriverjs.com/protractor-vs-webdriverio-vs-nightwatch/

Farhadi, F. 2018. Pros and cons of end to end testing tools. Luettavissa http://blog.scottlogic.com/2018/01/08/pros-cons-e2e-testing-tools.html