

Edward Freitas

# End-to-End-testauksen automatisointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotuotannon koulutusohjelma

Insinööriytyö

18.5.2018

Tekijä Otsikko	Edward Freitas End-to-End-testauksen automatisointi
Sivumäärä Aika	37 sivua 18.5.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Ohjelmistotuotannon koulutusohjelma
Ammatillinen pääaine	Tietotekniikka (ICT)
Ohjaajat	Yliopettaja Auvo Häkkinen
<p>Insinööriyössä selvitettiin, miksi ottaa End-to-End-testauksen automatisointia käyttöön projektissa. Miten siitä hyötyy, kun haluaa parantaa projektin laatua testeillä. Onko se kannattavaa ja hyödyllistä implementoida projektiin. Voiko ilman sitä pärjätä kilpailevan sovelluksen kehittämisessä.</p> <p>Työn tavoite oli selvittää, miksi End-to-End-testauksen automatisointia implementoidaan projekteihin. Onko sillä positiivista vaikutusta projektin valmistukseen, auttaako se kehitystä jollain tavalla vai nopeuttaako se jotain osaa projektin kehityksessä?</p> <p>Työssä käytettiin kirjoituspöytätyöskentelystä valmistukseen insinööriyön. Työssä opiskeltiin End-to-End-testauksen automatisoinnin toiminnallisuuksia, asennusta ja käyttökokemuksia, jonka jälkeen verrattiin omaan henkilökohtaiseen kokemukseen.</p> <p>Tulokseksi todettiin, että on oltava kehitysympäristön kanssa yhteensopiva kehikko End-to-End-testausta varten. Lisäksi End-to-End-testit ovat herkkiä antamaan virheellisiä ilmoituksia, jos sovelluksen ympäristö on epävakaa. Siitä on kuitenkin iloa muille organisaation jäsenille ja on myös kelvollinen projekti osien testaaja. On kuitenkin olemassa kevyempiä tapoja testata projektin osien toimintaa.</p> <p>End-to-End-testauksen automatisoinnin implementointi projektiin on melko kallista. Varsinainen asennus on helppoa, mutta pitkäaikainen käyttö on kallista. Olemassa on kaksi eri tapaa hoitaa toteutus, mutta molemmat tavat voivat olla yhtä kalliita vaihtoehtoja.</p> <p>End-to-End-testausta kannattaa toteuttaa, jos organisaatiolla on siihen varoja. End-to-End-testaus kykenee tunnistamaan web-sovelluksessa virheitä sekä samalla testaa liikkuvien osien toiminnallisuutta. End-to-End-testaukseen ei kannata keskittää kaikki organisaation testausresurssit vaan pitää sen yhtenä osana jatkuvassa integraatiossa.</p> <p>End-to-End-testauksen automatisointia ei ole pakko lisätä projektiin ja ilman sitä pystyy silti tekemään laadukkaan web-sovelluksen. Suositellaan kuitenkin ohjelman liikkuvien osien testausta, jonka voi suorittaa muilla testauskehikoilla.</p>	
Avainsanat	End-to-End-testauksen automatisointi, jatkuva integraatio, testausautomatisointikehikko, sovelluksen testausta

Author Title	Edward Freitas End-to-End test automation
Number of Pages Date	37 pages 18 May 2018
Degree	Bachelor of Engineering
Degree Programme	Software Engineering
Professional Major	ICT (Information and communications technology)
Instructors	Auvo Häkkinen, Principal Lecturer
<p>This thesis explores why End-to-End is used in projects and how it is used to ensure the quality of a web application. Is End-to-End testing worth the time and effort it takes to be implemented in the project? Can End-to-End testing be ignored entirely for the project's development or is it necessary for every web application?</p> <p>This thesis goes over why projects implement End-to-End testing and ponders on whether the testing has a positive or negative impact on a web application. This thesis is an exploratory research on End-to-End testing. It investigates its functionality, installation and other developers' user experience, and compares these with the author's personal experience in End-to-End testing.</p> <p>The results of the thesis seem to indicate that one should have a compatible framework before considering the implementation of End-to-End testing. The tests themselves are not very conclusive, so the environment itself will need to be stable for there to be useful results. There are faster and more efficient ways to test web application objects. End-to-End testing is, however, useful for other members of the development team, because it allows for visualization of tests when they are running.</p> <p>The big drawback for End-to-End testing is its high maintenance cost: installing a framework is straightforward and fast but maintaining tests on a continuous testing cycle becomes very expensive when hiring a developer for the creation of said End-to-End tests. It is possible to use keyword driven testing and have a non-developer performing End-to-End tests, but keywords will still need to be coded.</p> <p>In conclusion, End-to-End testing is worth implementing in a project if there are sufficient funds for it to be done. It is not essential for web applications, but End-to-End testing can identify errors through user experience, and it tests whether different web application parts are functioning correctly or not.</p> <p>End-to-End testing needs to be carefully balanced. It is not strictly necessary for it to be in a project, yet it is still recommended that a project has some way of testing a web application's moving parts.</p>	
Keywords	End-to-End testing, continuous testing, test automation framework, web application testing

# Sisällys

## Lyhenteet

1	Johdanto	1
2	End-to-End-testauksen automatisointi	2
2.1	Jatkuva integraatio	5
2.2	Toteutukset	8
2.2.1	Testauksen automatisoinnin pyramidi	8
2.2.2	Testauksen automatisoinnin jäätelötötterö	10
2.2.3	Testauksen automatisoinnintiimalasi	11
3	Web-sovelluksen perustoimintalogiikka ja testausmenetelmät	11
3.1	Asiakasohjelma (Client State)	12
3.2	Istunto (Session State)	12
3.3	Sovellustila (Application State)	13
3.4	Palvelintila (Server State)	13
3.5	Yksikkötestaus	14
3.6	Integroititestausta	15
3.7	Manuaalinen testaus	16
3.8	Testausautomatisointi	18
4	Protractor-testausautomatisointikehikko	20
4.1	Asennus	22
4.2	Protractorin käyttökokemus	23
5	Muut automaattitestaustehikot	24
5.1	TestCafe	24
5.2	Cypress.io	24
5.3	Nightwatch.JS	25
6	Muita testien automatisoinnin työvälineohjelmia	25
6.1	Näkemyks testausautomatisoinnista	25
6.2	Selenium	27
6.3	Katalon Studio	27

6.4	Watir	27
6.5	Rational Functional Tester	27
6.6	Robot Framework	28
6.7	Eggplant	28
6.8	Tricentis: API-testaaminen ja ylläpito End-to-End-testausautomaatiossa	29
6.9	Kattava automaatiotesti -työkalu	29
6.10	Qentinel	30
7	Analysointia	30
8	Yhteenveto	33
	Lähteet	35

## Lyhenteet ja terminologia

API	API (Application Programming Interface) on tietokoneohjelman välittäjä, jonka avulla kaksi sovellusta voivat keskustella keskenään.
Back-End	Back-End tarkoittaa koodia, joka ajetaan sivuston palvelimella, esimerkiksi yrityksesi palvelinhuoneessa tai pilvipalvelussa.
DevOps	DevOps on ketterä toimintamalli. Sillä pyritään parantamaan kehitysprosessia ohjelmistotuotantoprojekteissa.
End-to-End	End-to-End-testauksen automatisointi on testausprosessin kuvaustapa. Sillä halutaan testata sovelluksen toiminta alusta loppuun. End-to-End-testauksen automatisoinnissa testataan sovellusta asiakkaan perspektiivistä. Pyrkimyksenä on parantaa tuotteen laatua testien avulla.
Front-End	Front-end tarkoittaa kaikkea sitä koodia, joka ajetaan verkkoselaimessa käyttäjän edessä, kun ollaan sivuston kanssa vuorovaikutuksessa.
Jatkuva	integraatio Käytäntö, jolla pidetään kaikki toimintaympäristön järjestelmien tiedot yhteisessä pääjärjestelmässä.
Jatkuva kehitys	Termi kuvaa testausprosessia projektissa. Käytännössä testausta tehdään jatkuvasti jokaisen uuden version tultua ulos sovelluksesta. Samalla kun sovellus kehittyy, niin testit myös kehittyvät, tai niitä poistetaan sitä mukaan, kun sovelluksessa nähdään tarpeen muuttaa elementin toimivuutta.
JSON	(JavaScript Object Notation) Yksinkertainen avoimen standardin tiedostomuoto tiedonvälitykseen.
JavaScript	JavaScript on dynaaminen komentosarjakieli. Sillä voidaan lisätä Web-sivulle dynaamista toiminnallisuutta.

**Mock** Käytetään yksikkötestauksessa, tarkoituksena on simuloida objektin käyttäytymistä testauksessa. Hyöty on siinä, ettei tarvitse luoda epäkäytännöllisiä objekteja yksikkötesteihin testausta varten.

#### Mock-Palvelin

On avoimenlähdekoodin palvelin, jolla voi suorittaa Mock-testausta.

**Node.JS** Node.JS on sovelluskehys joka mahdollistaa JavaScriptin käytön palvelinpuolella.

**Protractor** Ohjelmistokomponenttikirjasto, joka tulee Node.JS:n mukana. Työkalua käytetään End-to-End-testeissä.

**Päätepiste** Päätepiste (endpoint) voi puskuroida lähetettyä ja vastaanotettua tietoa.

#### Regressiotestaus

Regressiotestaus on kaikentyyppistä testaamista. Tarkoituksena on ajaa vanhoja testejä läpi ja katsoa, ilmentyykö korjattuja virheitä.

**Reseed** Seed on todellinen satunnainen luku, jolla algoritmi lähtee generoimaan satunnaista numerosarjaa; reseed on todellisen satunnaisten numeron uudelleen luonti.

**Selenium** Selenium on testauskehikkotyökalu web-sovelluksille. Selenium on avoimenlähdekoodisovellus Apache 2.0 -lisenssissä.

**Skripti** Skripti on komentosarjakieli missä kirjoitetaan komentoja. Nämä automatisoivat tehtäviä, ilman varsinaista ohjelmointikieltä.

#### Yksikkötestaus

Testaamisen ja laadunvarmistuksen menetelmä, jossa lähdekoodin osat testataan yhdessä tai erikseen.

## 1 Johdanto

Kaikilla sovelluspuolen ohjelmilla on yleensä vikoja, joita pitää aina huolellisesti korjata. Ennen kuin niitä voi korjata, on ensiksi löydettävä vika itse sovelluksessa. Syöttökenttä, yhdistelmäruutu tai radiopainike on testattava, ja kaikki yhdistelmät on kokeiltava. Kun puhutaan End-to-End-testaamisesta, kyseessä on testien automatisointi, jollain halutaan simuloida testaamisprosessi käyttäjän näkökulmasta.

Ennen automaattitestausta yrityksissä palkataan tuhansia manuaalisia ohjelmistotestaa- jia ohjelman tuotekehityksen rinnalle putsamaan rippeitä yksikkötestauksesta. Manuaalinen testaus on itseään toistavaa työtä ja inhimillisiä virheitä tulee helposti. Varsinkin, kun virheitä on tullut putsattua useamman kerran ohjelmasta. Yhtä bugia voi joutua testaamaan useamman kerran ennen kuin se edes varmennetaan virheeksi sovelluksessa.

Kyseessä on kirjoituspöytä tutkimus, jossa pyritään selvittämään End-to-End-testaus automatisoinnin tarpeen. Miksi on tarpeellista olla End-to-End-testausta, miten se on hyödyllistä ja onko kannattava lisätä sitä projektiin? Yksikkötestaaminen on valtaosaisesti tehokas tapa tarkistaa, että sovellus tosiaan toimii virheettömästi. Isoin valitus End-to-End-testauksessa on sen hitaus, sekä virheelliset esiintymät aina, kun lähdetään ajamaan uusinta koontiversiota sovelluksesta.

Työn tavoite on selvittää automaattitestauksen merkitys yrityksessä, kannattavuus, syyt olemassaololle, rahallinen arvo, hyvät käytännöt sekä muut laadun varmistukseen liittyvät syyt. Myös eriäviä näkökulmia automaattitestauksen toteutuksessa halutaan tutkia. Työssä otetaan huomioon ohjelmistokehittäjien ja manuaalitestaa- jien eri näkemykset End-to-End-testauksen hyödystä projektiin.

Tässä työssä käydään läpi, mitä End-to-End-testaaminen on ja miten se on mukana jatkuvassa integraatioprosessissa. Tutustutaan Protractorin kehikkoon, koska yritys käytti Angularia sovellusalustana projektin alkaessa. Protractorin avulla voi suorittaa End-to-End-testejä. Työssä käydään läpi muita testien automatisoinnin työvälineohjelmia ja ideoita suorittaa testausautomaatiota asiakkaan näkökulmasta.



## 2 End-to-End-testauksen automatisointi

Tutustumme End-to-End-testauksen automatisointiin. Luvussa kerrotaan, mitä End-to-End-testaus on, mihin sitä tarvitaan, käyttökokemuksia sekä asennusprosessia.

End-to-End-testauksen automatisointi on testausprosessin kuvaustapa. Sillä halutaan testata sovelluksen toiminta alusta loppuun. Tarkoituksena on parantaa tuotteen laatua testien avulla. End-to-End-testauksessa halutaan testata koko ohjelman toimintaa asiakkaan perspektiivistä. Pyrkimyksenä on simuloida asiakkaan käyttäytymistä sovelluksessa niin tarkasti kuin vain on mahdollista. (2.)

End-to-End-testaaminen voi olla hyvin haastavaa, sillä on hyvin monta seikkaa, jotka voivat mennä pieleen testien ajoissa. Yleensä se lievittää päänsärkyä asiakkailta, koska he eivät törmää virheisiin ohjelman käytössä. Seuraavana on lista asioita, jotka menevät pieleen End-to-End-testien implementaatioissa. (1.)

- Projektin kehitysyksikkö on myöhässä työaikataulusta.
- Virheen löytäminen epäonnistuneelle testille on työläs.
- Virheelliset esiintymät pilaavat todenmukaisen tulokset.
- Löytyy isojen virheiden takaa, paljon pieniä virheitä.

Ajoitusviat ovat suurimpia ongelmia, mitä löytyy. Nämä ongelmat yleensä korjataan ”odotusfunktiolla”, jossa yleensä viitataan elementin paikalla oloon (1).

Jos elementti ei ole vielä ilmentynyt näkymään pitkien latausaikojen takia, funktioon on lisätty metodi, joka on sisään rakennettu kehikkoon, kuten Protractorin (End-to-End-testausautomaatiokehikko). Protractorissa voi myös jäädyttää testejä tietyissä tilanteissa. Tilanteen jäädytys lisää odotusaikaa testin ajamisessa, joten kannattaa säästellä niiden käytössä. Parempi ratkaisu olisi antaa testausohjelman tietää, että ohjelma suorittaa taustalla tapahtuvia latauksia.

Node.JS on avoimen lähdekoodin ympäristö, joka edesauttaa JavaScript-koodin suorittamista palvelimella. Node.JS on pakko asentaa, jos haluaa käyttää Protractor-kehikkoa testauksen automatisoinnissa.

End-to-End-testauksessa on koottava sovellus testiympäristössä, jotta sovellusta voidaan testata (1). Kun projektissa muutetaan jonkin komponentin toimintaa, komponenttiin liitetty testi on myös muokattava. Tyypillisesti testaajan työt loppuvat sen jälkeen, kun virhe on löytynyt. Sitten testaajat ilmoittavat virheen olemassaolon, jonka jälkeen ohjelmistokehittäjät korjaavat virheen.

Tietokannan hidas datan päivittyminen sovellukselle tuo ongelmia tuotantoon. Vaikka sovellus olisi latautunut selaimessa, tietokannasta lähtenyttä dataa ei ole vielä ehtinyt päivittyä selaimelle. Syntyy omituisia virhe tilanteita, missä käyttäjän pakolliset syöttökentät ovat tyhjiä kuten nimi, sukunimi, puhelin numero ja sähköposti. Protractorin on odotettava, kunnes nämä kentät täyttyvät, ennen kuin se julkaisee virheen. Paras tapa, jolla estetään virhe, on käyttää metodia, joka tiedottaa Protractorille ohjelman suorittamista taustalla tapahtuvista latauksista. Metodi löytyy Protractorin sisään rakennetusta kirjastosta, jossa on paljon hyödyllisiä metodeja ja funktiota, jotka parantavat ohjelman toiminnallisuutta. Protractorin kirjasto myös toimii sovelluksissa, jotka eivät käytä Angularia suoraan. Protractorin nopeutta testien ajossa ei voi hidastaa asetuksessa. (1.)

Kun halutaan nopeaa palautetta testiajoista, on suoritettava testit Mock-palvelimessa. Mock-palvelin on avoimen lähdekoodin palvelinkehikko, jossa voi suorittaa Mock-testejä. Mock-testit ovat testejä, jossa simuloidaan objektin käyttäytymistä testauksessa. Hyöty on siinä, ettei tarvitse luoda epäkäytännöllisiä objekteja yksikkötesteihin testausta varten. Mock-palvelimen avulla löytyy tyydyttävä määrä virheitä. Se ei käyttäydy kuin käyttäjä testien ajossa. Se ei tunnista vikoja, joihin käyttäjä voi törmätä. Käyttäjä ottaa vastaan API-kutsuja aina, kun hän on jollain tavalla vuorovaikutuksessa ohjelman kanssa. (1.)

Mock-palvelintoteutuksen voi tehdä myös API-kutsujen nappaamisen pohjalta, missä testit odottavat fyysistä muutosta sovelluksessa. API-kutsujen nappaaminen tulee hidastamaan integraatiotestausten nopeutta. Testien etuna on se, että ne simuloivat paremmin todellisuutta. Kuitenkin, jos on mahdollista, olisi hyvä käyttää End-to-End-testausautomatisointia sekä Mock-palvelimen testausta projektissa. (1.)

Jokaista uutta muutosta on kuitenkin testattava integraatiohaarassa, eikä ohjelmistokehittäjä voi odottaa 50 minuuttia koontiversion valmistumiseen. On suositeltava pitää End-to-End-testausautomatisointia erillään integraatiotesteistä (1). Ohjelmistokehittäjä tarvitsee Mock-palvelimen, jotta hän voi kaikessa rauhassa tehdä muutoksia ohjelmaan.

Manuaalitestaaaja ja laadunvarmistaja haluavat mahdollisimman todellisen kuvan ohjelman toiminnallisuudesta, joten testien on vastaan otettava API-kutsut ja verrattava niitä palvelimen dataan. End-to-End-testejä luodaan kattavuutta varten, sillä yksikkö- ja integraatiotestit sijaitsevat usein pienessä osassa ohjelmaa, eivätkä ne yksinkertaisesti riitä varmistamaan tuotteen laadukkuutta. (2.)

Jos halutaan todellista End-to-End päästä päähän -testausta, on testit ajettava varsinaisesta palvelimesta, jota tullaan käyttämään (2). Ohjelmistokehittäjä väittää vastaan, koska työnteko tulee pidentymään huomattavasti. Jokaisen liitospyynnön jälkeen tehdään koontiversio, vaikka muutos tapahtui yhdessä komponentissa. Aina kun tehdään muutoksia, on ajettava uusin koontiversio, jossa samalla testataan yksikkötestejä ja End-to-End-testejä (1).

Mock-palvelin on tarpeeksi realistinen alusta, jossa testejä voidaan ajaa ja ongelmia voidaan poimia sekä korjata (1).

End-to-End-testaamisen idea on tarkistaa elementtien toimivuus sekä ohjelman toiminta. Tällä varmistetaan, ettei integrointivaiheessa synny ongelmia sovellukselle. Virheitä voidaan napata myöhemmin ajamalla koko järjestelmätestiä. Päivän päätteeksi ajetaan End-to-End-testit, jotta aamulla nähdä mahdollisia virheitä edeltävän päivän työstä (1).

Priorisointi on tärkeää, sillä projektin kehityksessä ei haluta hukata liikaa aikaa testien luomisessa, jotka eivät edistä ohjelman kehitystä. Suositellaan voimakkaasti, että testitapaukset keskitetään pääosin regressiotehtävien tarkistukseen. Regressio on vaativa ja pitkästyttävää prosessi, jossa vaaditaan testaajalta kärsivällisyyttä sekä tarkkuutta suorittaa tehtävä toistuvasti uudelleen. Koska ihminen helposti tekee virheitä tuloksena, saattaa ilmetä virheellisiä esiintymiä, joiden tarkistamisessa kuluu paljon aikaa. (2.)

Testien integrointi tuotantoon on tärkeää. Yritys haluaa varmistaa tuotteen laadukkuuden ajamalla integraatiotestejä aina, kun pusketaan uusinta koodia tuotantoon (2). End-to-

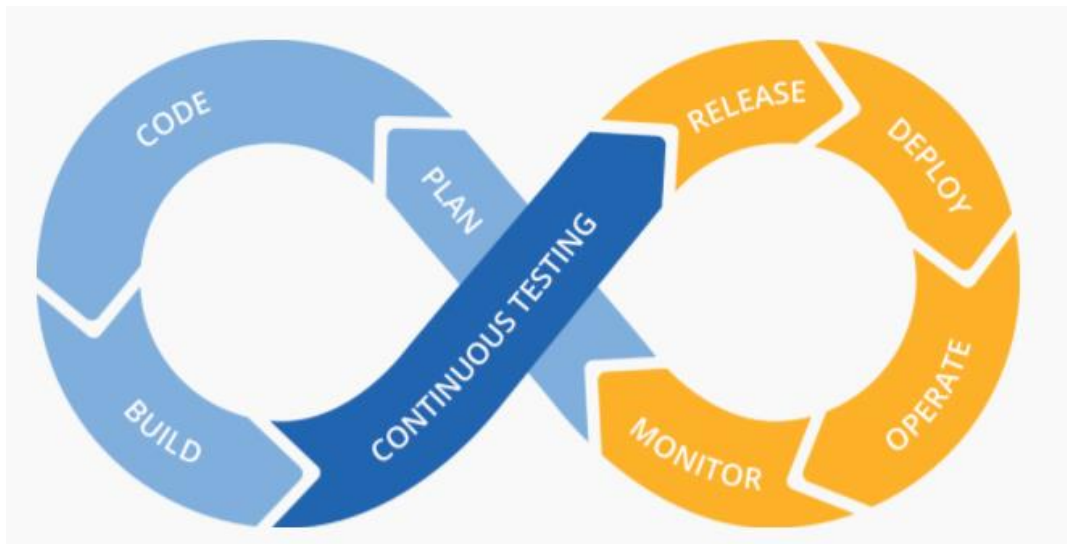
End-testit, on lisättävä ohjelman integraatiovaiheeseen, jotta voidaan estää virheiden ilmentymä ohjelmassa. Manuaalitestaaajat voivat ajaa End-to-End-testejä, koska tahansa, jotta he voivat tarkistaa, ettei tuotantoon ole syntynyt virheitä. Kun sovellusta kehitetään eteenpäin ja implementoidaan suuria lomakkeita sovellukseen, on manuaalitestaaajia varten kehitettävä automatisoituja regressiotestejä (1).

Automaattitestauksen yhtenä tavoitteena on helpottaa regressiotestausta, jotta edistetään ohjelman kehitystä sekä vähennetään manuaalitestauksen tarvetta projekteissa. Tällä hetkellä automaattitestausta ei omillaan takaa ohjelman laatua eikä korvaa manuaalitestaaajien tarvetta. Ohjelman laadun testaus on yhä hyvin kallista ja vie kehitykseltä paljon aikaa eikä testausautomaatio muuta tätä ongelmaa.

Lopputuloksena jokainen ohjelmistokehittäjä joutuu aina koodimuutoksen jälkeen päivittämään testejä muokatussa komponentissa. Tämä vie aikaa tuotannolta, mutta se parantaa tuotteen laadukkuutta nappaamalla virheet ajoissa eikä vasta ohjelman julkaisun jälkeen. End-to-End-testaus pääsee melko lähelle vastaamaan todellista käyttäjää ohjelman käytössä. Pitkällä juoksulla automaattitestausta maksaa itsensä takaisin hyvän asiakaspalautteen kautta. (2.)

## 2.1 Jatkuva integraatio

Kuvassa 1 esitetään jatkuva integrointi ohjelmistokehitysprosessi, jonka tavoitteena on pitää sovellus puhtaana virheistä. Ideana on havaita ongelmia ohjelman ajoissa, jotta niihin voi puuttua ja korjata. Kuvassa 1 havaitaan, kuinka jatkuva integraatioprosessi on ohjelman tuotannon mukana ikuisesti.



Kuva 1 Jatkuvan integraation prosessi (4)

Alustavasti hyväksytään, että jatkuva integraatio malli otetaan käyttöön. Sitten luodaan kehitysympäristö, käyttöä ja testausta varten (release, deploy). Ohjelmaa suoritetaan jatkuvasti (operate), jolloin käytön aikana tulee esiin virheitä. Ohjelmaa samalla valvotaan hyvin tarkasti (monitor). Kun virheet löydetään, suunnitellaan (plan) parasta tapaa paikata virheitä ja luodaan sitä varten jokin turvaverkko mahdollisesti toistuvasta virheestä, kun projektia kehitetään. Kehitys jatkuu (code), luodaan komponentteja, niihin lisätään vastaavia testejä ja sitten ohjelma kootaan uusimmasta projektiversiosta (build). (4.)

Yritykselle asiakastyytyväisyys on valuutta markkinoilla. Tunnettu tapa parantaa tyytyväisyyttä on vähentää riskejä. Vaikka kyseessä olisi kuinka lahjakas ohjelmistokehitystiimi, virheitä tulee ja virheet ovat riski, jonka kanssa yritys tulee aina painimaan.

Jatkuvan integraation perusoppi on testata ajoissa, nopeammin, usein sekä automatisoida testausprosessia. Ajoistatestaaminen vaatii, että ympäristö on kunnossa sekä sen varmistamista, että ennen kuin lisätään uusia muutoksia tuotantoon, etteivät muutokset johda virheisiin. Nopeassa testaamisessa pyritään tiputtamaan ohjelmistokehittäjältä mennyttä aikaa testien ajossa. Testien ajaminen niin usein kuin mahdollista on elintärkeää ohjelman laadun kannalta. Kun jotain automatisoidaan, yleensä sen seurauksena tehtävä helpottuu. Pyrkimyksenä automatisoinnissa testien kannalta on helpottaa ihmisten ymmärrystä ohjelmointikielestä avainsanoilla sekä helpottaa regressiotestausta mutkikkaiden sovellusten kohdalla. (3.)

Jatkuva integraatio pyrkii lisäämään End-to-End-testauksen automatisointia. Tällä parannetaan tuotteen laadunvarmistusta (4). Automatisointi nopeuttaa testausprosessia. Sen seurauksena tuotteen kehitys edistyy nopeammin ilman, että uhrataan tuotteen laadukkuutta.

Kun ohjelma saapuu tuotantovaiheen (pre-production), laadunvarmistusyksikkö kykenee hyödyntämään ohjelmaan luotuja yksikkö- ja End-to-End-testejä. Laadunvarmistusyksikkö tulee päämääräisesti keskittymään virheiden löytämiseen sekä testien ylläpitoon. (3.)

Keskeiset perustekijät jatkuvassa integraatiossa ovat riskienarviointi, yrityksen sisäinen politiikka, määritetyt tavoitteet, analysointi, testien optimointi ja palvelun virtualisointi (3).

Riskienarviointiin kuuluu tehtävien lievennys, jonka tarkoituksena on luopua tehtävistä, jotka ovat tarpeettomia tai vievät liikaa aikaa tuotantotiimiltä. Tekniset velat ovat suunniteltuja aikatauluja jotka ovat onnistuneet venymään. Laadun arviointi missä mitataan testeillä ohjelman kattavuus, eli kuinka monta komponenttia on testattu ohjelmassa (3).

Yrityksen sisäinen politiikka syntyy silloin, kun halutaan maksimoida tuottoa liikemaailmassa. Yritys ottaa vastaan käytännöt, joilla varmistaa yrityksen kasvua ja olemassaoloa (3). Yritys pitää tuotantotiimin nähden viimeisen sanan päätöksenteossa.

Määritetyt tavoitteet ovat läpipääsykriteeri tuotteen minimitoiminnallisuudelle (3). Ajatuksena on saatava luotettava tasapainoinen työympäristö, jonka ympärillä voi rakentaa projektia. Nämä tavoitteet ohjelman toiminnallisuudelle ovat yrityskohtaisia määritteitä.

Virheen analysoinnin tarkoituksena on välttää toistuvia virheitä sekä automatisoida staattista koodia sekä tehostaa tuotantoa (3).

Testien optimoinnissa halutaan ylläpitää yksikkö- ja automaattitestejä sekä kehittää niitä jatkuvasti sovelluskehityksen rinnalla. Samalla halutaan myös parantaa testi kattavuutta sovelluksessa. (3.)

Palvelun virtualisoinnissa halutaan puskea prototyypiversio ohjelmasta visualiseen maailmaan, jotta tuotteen laatua voitaisiin parantaa (3).

## 2.2 Toteutukset

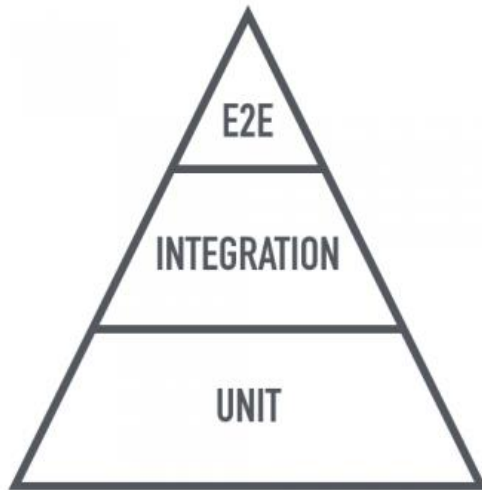
On monta tapaa lähteä toteuttamaan projektia. Kun idea syntyy, yleensä suunnitellaan toteutusta ennen kuin lähdetään kehittämään projektia. Kyseessä voi olla siis hyvin pitkä ja työläs prosessi, jossa mahdollisesti idea hylätään kokonaan.

Ohjelmoinnissa mietitään enemmän sovelluksen kehittämistä ja ohjelman arkkitehtuurista rakennetta. Myös mietitään enemmän tuotteelle oikeaa käyttöä. Mutta harvoin mietitään tuotteen laatua. Jotta ohjelmistokehittäjän työ helpottuisi on annettava selkeät määritteet sovelluksen kehittämistä varten. Tuote ei ole olemassa ennen kuin sitä voidaan koota ja käyttää tarkoitustaan varten. Kaikki mikä seuraa sen jälkeen, voi lisätä tai jättää.

Lisensointi on yleistä näissä työkaluissa, mikä tarkoittaa sitä, että ne toimivat vain tiettyjen ohjelmien avustuksella. Joka luo todella sotkuisen polun tuotteen valmistukselle. (5.)

### 2.2.1 Testauksen automatisoinnin pyramidi

Pyramidimalli (kuva 2) on vakain ja yksinkertaisin malli, jota suositellaan voimakkaasti firmoille, jotka ovat aloittamassa End-to-End-testausautomatisoinnin implementointia projektiin. Näemme stabiilin ja vakaan yksikkötestien olevan pohja rakeenteelle, ainakin laadunvarmistus tiimin näkökulmasta. (5.) Mallin kriitikot eivät pidä siitä, koska se lisää ylimääräistä työtä ja hidastaa tuotantoa. Se myös puskee ideaa, missä pitää muokata tiimin rakennetta, jotta saadaan pyramidi toteutettua. (7.)



Kuva 2 Testauksen automatisoinnin pyramidi (5)

Yksikkötestit (unit) ajetaan koonnissa nopeasti ja saadaan selkeä palaute projektin toimivuudelle (5). Sekä testien luonti että muokkaus ovat yleensä helppoa ja vaivatonta. Kun puhutaan yksikkötestistä, kyseessä on yksi tai useampi testi, jossa testataan yhtä luokkaa tai yhtä metodia (8). Seuraavassa vaiheessa uusi testattu elementti liitetään (integroidaan) osaksi aiempaa kokonaisuutta, jossa sitten ajetaan kaikki ohjelman yksikkötestit. Ideana on varmistaa koonnissa, että muutokset eivät vaikuta muihin komponentteihin tai hajota ohjelmaa jollain tapaa. (9.) Viimeisenä on End-to-End, missä pyritään luomaan todenmukainen simulaatio käyttäjän kokemuksesta ohjelman käytössä (5).

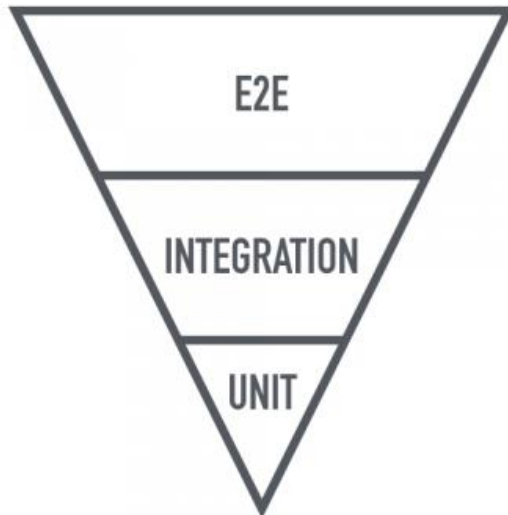
Kun katsotaan mallia, huomataan, että End-to-End on jätetty melko pieneksi osaksi pyramidia. Tämä johtuu siitä, kuinka hidaskäyttötestien ajaminen on ja myös testien luomiseen kulunut aika. (5.) Testien toiminta on varmistettava ennen kuin ne lisätään integrointiprosessiin. Suurempi haitta on koonti itsessään, missä testit ajetaan reaaliajassa. Testit ovat myös epävakaita ja tuottavat usein virheellistä esiintymää (7). Tämän vuoksi vain kriittiset päätepisteet testataan, jotta ohjelmistokehittäjät saavat rauhassa kehittää ohjelmaa eteenpäin (5). Kuvasta puuttuu manuaalinen testaus, mutta yleensä pienyrityksillä ei löydy heti varoja palkata testaaajia. Käytäntönä on yleensä pyrkiä hankkimaan vapaaehtoisia henkilöitä testaamaan ohjelmaa.

Pyramidimallilla halutaan korostaa yksikkötestien tärkeys. Yksikkötestit ovat nopeampia kuin End-to-End-testit ja mallin tarkoituksena on välttää projektin kehityksen hidastamista.



### 2.2.2 Testauksen automatisoinnin jäätelötötterö

Jäätelötötterömalli (kuva 3) on pyramidimalli, jossa projektin tuotantotiimi unohti yksikkötestien merkityksen ohjelmanlaadun parannukseen. Kolikon toisella puolella pyramidi on käännetty ylösalaisin ja päälle lisätään manuaalista testausta. (5.) Tämä malliin ideana on herättää kritiikkiä projektituotannon tiimille. End-to-End-testaamisen ongelma on yleensä tietää, mihin lopettaa komponenttien testauksessa.



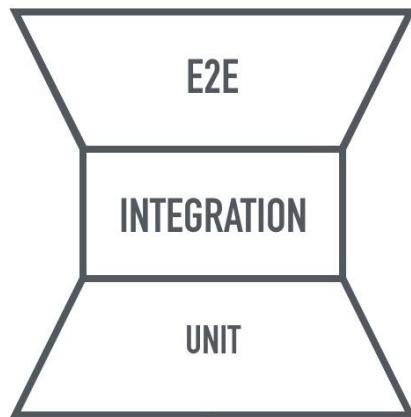
**Kuva 3 Testauksen automatisoinnin jäätelötötterö, vastakohta pyramidimallista sekä huono toteutus (5)**

Yksikkötestejä on vähennetty huomattavasti, mistä seuraa huomattava supistus integraation luotettavuudesta. Tässä kuvassa ongelmana on testien laadukkuus. Yksikkötestausta tehdään huomattavasti vähemmän. Projektin kasaamisessa menee enemmän aikaa, ja kun huomataan, että komponenttia on muokattava, projektin kehityksessä menee enemmän aikaa (7). Manuaalitestaus on tässä mukana, koska manuaalitestajat ajavat skriptin, joka sitten ajaa testit. Se johtaa End-to-End-testien lisäämiseen, jolla helpotetaan manuaalitestajien työtä (1). Näitä testejä kuitenkin pitää ohjelmoida ja siihen tarvitaan ohjelmistokehittäjän osaamista (6).

Jäätelötötterömallin isoja ongelmia ovat pitkät läpimenoajat, heikko laatu, arvaamattomat tulokset, virheet menevät tuotantoon ja resurssit kuluvat hukkaan (7).

### 2.2.3 Testauksen automatisoinnitiimalasi

Tiimalasimallissa (kuva 4) on pyritty tasaamaan End-to-End-testit yksikkötestien kanssa (5). Tarkoitus on luoda tasapaino ja käyttää integraatiotestausta pullonkaulana ohjelman luotettavuudelle.



Kuva 4 Testauksen automatisoinnin tiimalasi (5)

Mallin hyvä puoli on mahdolliset käyttötarkoitukset stressitesteissä, sillä testejä voidaan suorittaa erilaisilla yhdistelmillä samaan aikaan. On mahdollista luoda jonkinlainen stressitestialusta. Vaikkakin stressitestausta voidaan suorittaa klassisella ohjelman resurssien ylikäytöllä, testauksen automatisointi antaa mahdollisuuden kehittäjien nähdä, mitä kaikkea ohjelmassa menee pilalle, eikä pelkästään kuormituspalkkeja ohjelman suoritukselle. (5.)

Yleinen kritiikki on, että End-to-End testejä ei kannata olla paljon, sillä se hidastaa ohjelman koontia. Miksi vaivautua testaamaan ohjelman toiminnallisuutta erilaisilla yhdistelmillä, kun sitä jo voidaan tehdä kevyemmin ja nopeammin yksikkötestien avulla? Sama valitus pyramidimallista on tuotantotiimiä pitää rakentaa mallin ympärille, eikä soviteta käytäntöä tuotantoon.

## 3 Web-sovelluksen perustoimintalogiikka ja testausmenetelmät

Luvussa tutustutaan web-sovelluksen toiminta logiikkaan sekä useampaan testausmenetelmään. Kyseisessä End-to-End-testausautomatisointitoimintalogiikassa käytettiin

Angularia, Protractoria, Railsia ja Node.JS:ää. Angular on Googlen käyttämä web-sovel-luslusta (application platform). Angular mahdollistaa sovelluksen ohjelmointia TypeSc-ript-kielillä. TypeScript on Microsoftin ylläpitämä ohjelmointikieli. Protractor on End-to-End-testausautomatisoinnin kehikko. Protractorin avulla voidaan luoda automaattites-tejä. Rails on MVC-arkkitehtuuri. Sillä erotetaan käyttöliittymä sovellusalueidosta. Node.JS on JavaScript runtime -ympäristö. Node.JS tekee mahdolliseksi suorittaa koo-dia suoraan palvelimella.

### 3.1 Asiakasohjelma (Client State)

Palvelimen tila ilmoittaa, missä tilassa on tämänhetkinen palvelin ja sen rakenne. Oleel-liset osat ovat käyttöjärjestelmä, selain, keksit, paikallinen tietokonemuisti ja selaimen lisäosat. Selenium (on testaus kehikko web-sovelluksille) antaa elementeille tunnistheet, jotka sitten Protractor kykenee tunnistamaan ja ajamaan testitapauksissa. Palvelimen voi optimoida ajamaan automaattitestejä useammassa ikkunassa käyttäen eri selaimia. (18.)

### 3.2 Istunto (Session State)

Istunto on kategorian alapuolella asiakasohjelmasta. Istunto ilmoittaa, millä tunnuksella on kirjauduttu sisään. Sen avulla voi autentikoida sisäänkirjautumista. Istunnon avulla voi sivuuttaa sisäänkirjautumisen, joka auttaa automaatiotestauksessa, koska sisäänkir-jautumista ei tarvitse aina testata, kun ajaa End-to-End-testejä sovellukseen. Voidaan tehdä itse käyttäjätestejä, esimerkiksi keillä käyttäjillä on mitäkin oikeuksia sovelluk-sessa. Angularilla on metodi, jolla saa asetettua sähköpostiautentikointitestejä toiminta-kuntoon. On myös mahdollista asettaa valtakirja autentikointia varten. Kun käyttäjä saa-puu sivulle, hänelle määritetään valtakirja, joka on vieraspassi kyseisellä sivulla. Valta-kirja mitätöityy, kun käyttäjä astuu ulos sivulta tai valtakirja vanhenee. Valtakirja avulla pystytään tallentamaan käyttäjän tekemät muutokset ilman sisäänkirjautumista järjestel-mään. (18.)

### 3.3 Sovellustila (Application State)

Sovellustila tiedostaa, missä osassa sovellusta ollaan. Kuten esimerkiksi hakemisto, mistä on pääsy muihin sovelluksen osiin. Hyvä käytäntö on pitää jokaiselle tilalle oma URL-linkki. URL-linkeillä päästään luomaan tila uudestaan. Valitettavasti aivan kaikkea ei voi tunkea URL-osoitteeseen. Tietyt elementit, kuten alasetolistat, jotka luovat ponnahdusikkunan auki aktivoinnissa, ei tule rekisteröimään mitään muutosta URL-linkkiin. Tämä tarkoittaa sitä, että nyt tallennetun URL-osoitteen avulla pääset takaisin samaan tilaan selaimessa, mutta elementti muutokset, kuten alasetolistat, eivät tule tallentumaan palatessa sivustoon. (18.)

End-to-End testaamisessa on oleellista saada luotua itselleen tila sovelluksessa, jotta vältetään testin toistoa (18). Vaikkakin elementin uudelleentestaaminen olisikin hyvä tapa varmistaa laatua ohjelmassa, sen jatkuva uudelleentestaaminen hidastaa ohjelman koontia ja tekee testeistä toisistaan riippuvaisia. Tämä aiheuttaa ylimääräistä työtä ohjelmistokehittäjälle, sillä virhe epäonnistuneessa koonnissa on hankalalukuista. Halutaan saada selville, mikä hajosi, eikä vain virheilmoitusta sadoista epäonnistuneista testeistä.

### 3.4 Palvelintila (Server State)

Palvelimella on yleensä useampia komponentteja, jotka voivat olla eri tiloissa. Lisäksi palvelimella on relaatiotietokanta, joka seuraa relaatiomallia. Avainsanavarasto (key value store) on tietovarasto. Välimuisti (in-memory cache) on muistia, jonka tehtävänä on nopeuttaa tietokoneen toimintaa. Oheislaitteisto tiedostovarasto (peripheral file stores) kuten S3. S3 on Amazonin Simple Storage Service, joka tekee tietojenkäsittelystä helpomman ohjelmistokehittäjille. Paikallistiedosto välimuisti (local file cache) tarkoituksena on nopeuttaa verkko tiedostoihin pääsemisen sovelluksessa. (18.)

Valitettavasti testattavia komponentteja ei voi luoda uudestaan testaamista varten. Luontivaiheessa tulee virheitä, jotka eivät välttämättä edusta todenmukaista käyttäjän löytämää virhettä. Mockin avulla voi luoda testattavia objekteja eli ohjelman osia, joita pitää testata sovelluksessa. Hyvä käytäntö Mockin käytössä on pitää se yksinkertaisena, ei ylimääräistä tekstiä, vaan selkeitä avainsanoja, jotka osoittavat ongelman. (18.)

Haaste on yleensä saada varmistettua, että datan tila ei muutu testiajajen aikana (18). Eli jos etsimme henkilöä Pekka hakukoneesta eikä häntä sitten löydykään enää datasta, törmäämme testiajossa virhetilaan, jota ei aiheuttanut mikään virhe, vaan testi oli riippuvainen edellisestä. Edeltävän testin tarkoitus oli luoda Pekka järjestelmään, ja testi joko tiputettiin tai se epäonnistui.

Hyvä käytäntö on pitää vakituista testidataa tietokannassa. Se säilyy varmuuskopiona, kun halutaan nollata palvelin. Palvelin kannattaa aina palauttaa alkutilaan jokaisessa testikoontiajossa. Muuten testit epäonnistuvat, koska data on jo kertaalleen syötetty tai pahempaa palvelimeen luodaan jatkuvasti samaa dataa. Aivan kaikkea kuitenkin ei voida pitää staattisena datana, jota sitten testataan. Merkkiaikaleimat, valtakirjat ja tunnistimet, joita luodaan satunnaisesti tai automaattisesti, on vaikea tarkistaa, koska data on yleensä muuttuvaa. (18.)

Jos testeihin kaivataan lisää dataa, muokkaa esiasteita kannan palautuksessa. Reseed tarkoittaa, että tietokannasta generoidaan eri algoritmin avulla uusia esiasteita. Tämä tehdään kutsumalla haluttu päätepiste (Endpoint), jossa uudelleengenerointiprosessi käynnistyy. (18.)

Aina kun ajetaan kanta uusiksi, skripti ajaa tehtäviä seuraavassa järjestyksessä. Tuhoaa ja luo uudelleen tietokannan, ettei data kopioi itseään. Generoi uudella algoritmilla tietokantaa ja luo variaatiota virheiden löytämiseen. Lisää määritetyt esiasteet, joita voi muokata tarpeen mukaan. Skripti tyhjentää sekä uudelleen alustaa osoittimet ja hakukoneen. Skripti tyhjentää istunnon aikana syntynyttä dataa estääkseen datan monistumisen. (18.)

On myös mahdollista palauttaa kanta lisäämällä tietokanta suoraan sovellukseen tai käyttämällä Shell-skriptiä, jossa voi tallentaa usein käytettyjä komentosarjoja tiedostoon. Niitä voi käynnistää kutsumalla skriptin nimeä, jonka jälkeen se suorittaa siinä olleet komennot. (18.)

### 3.5 Yksikkötestaus

Yksikkötestauksessa testataan, että elementti käyttäytyy sekä toimii oikein ja että se tuottaa oikeat arvot. Karma on työkalu, joka tulee Angularin mukana. Se luo nopeasti

selaimelle tyhjän HTML-sivun, joka täyttyy elementeillä, joita sovellus käyttää. Yksikkötestauksessa testataan pientä palaa koodista, missä mitataan funktion tai luokan käyttäytymistä. Tarkoitus on varmistaa kyseisen funktion tai luokan toiminnallisuus. Yksikkötestauksessa käytetään myös kehikoita, ajureita (pala koodia joka käynnistää testit), Mock (tarkoituksena on simuloida objektin käyttäytymistä testauksessa) ja jäsenfunktiota, joka on olioon määritelty funktio. (8.)

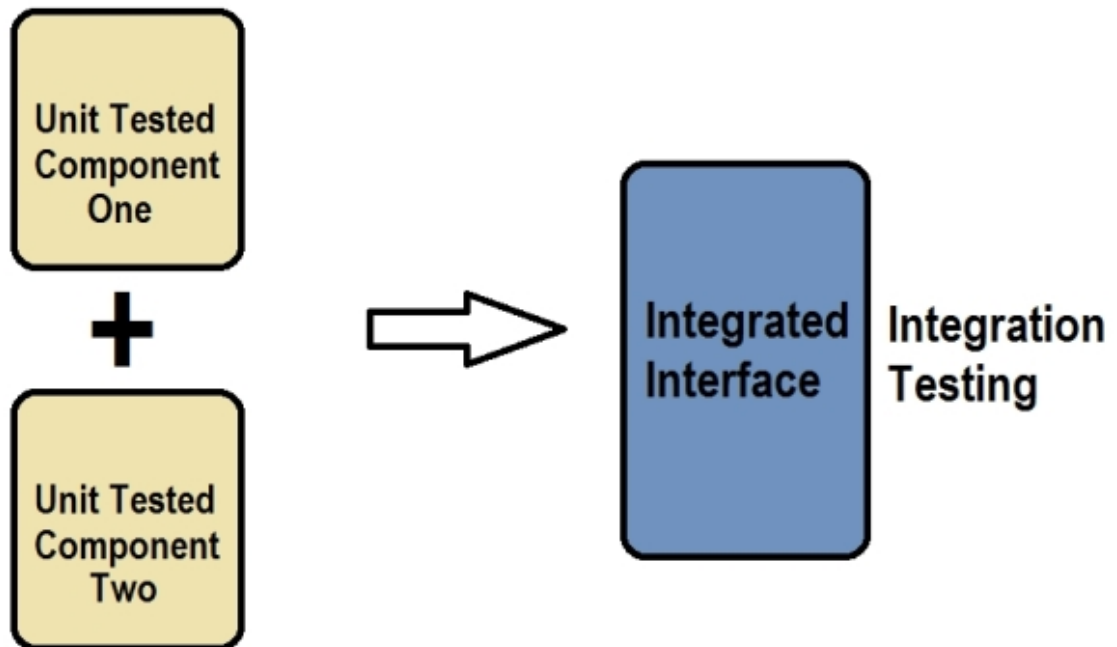
Yksikkötestauksen suoritusta varten on löydettävä yhteensopiva kehitysympäristö ja testauskehikko. On vältettävä luomasta testauskomponentteja jokaista toimintoa varten. Se hidastaa testien ajoa ja koontiprosessia. Se luo testauskomponentteja niissä toiminnissa, missä raskaasti vaikutetaan ohjelman käyttäytymiseen, josta syntyy virheitä usein. Eristetään kehitysympäristö testaus ympäristöstä, jotta koonti ei palauttaisi virheellistä esiintymää. Verrataan testidataa tuotantoon, jos vaikka tuotantoon on onnistunut piiloutumaan virheitä. Kirjoitetaan testejä, jossa paljastetaan virhe. Se helpottaa virheen korjausta, jos ja kun se toistuu. On kirjoitettava testejä, jotka ovat riippumattomia toisistaan, niin ettei synny domino vaikutusta testeihin. On oltava kattavia testejä, toistorakenteen testauksessa. Silloin kun on kyseessä regressiotestausta, missä testataan lomakkeita. Käytetään versiohallintaa, jolla voi pitää kirjaa tiedostoon tehdyistä muutoksista. On kirjoitettava testitapauksia, missä taataan testattavan koodin laadukkuuden ja toiminnallisuuden. Aina kun koontiversio on ajossa, samalla myös testataan kaikki oleelliset testit. On suositeltavaa pitää End-to-End-testejä erillään koontiversiosta. (8.)

### 3.6 Integroititestausta

Integraatiotestauksen tarkoitus on testata järjestelmän eri osia, jotta tiedetään, miten sovellus käyttäytyy. Integraatiotesteistä on hyötyä silloin, kun yksikkötestit eivät riitä todentamaan kahden eri järjestelmän toiminnallisuutta. Testit missä halutaan testata sovellusta hakemalla tietokannasta jotain tallennettua dataa. (10.)

Lisätty monimutkisuus integraatiotesteissä tekee siitä hitaamman kuin yksikkötestauksen. Lisäksi niitä pitää erikseen sovittaa järjestelmään sekä on luotava testi dataa, jotta saadaan tarkkaa tietoa sovelluksen toiminnasta. (10.)

Kuvan 5 mukaan yksikkötestikomponentti plus toinen yksikkötesti komponentti muodostavat integraatorajapinnan. Tämä rajapinta muodostaa pohjan integrointitesteille, eli se on kokoelma useista yksikkötesteistä.



Kuva 5 Integraatiossa testataan useita yksikkötestejä (9)

Jatkuvan integraation pohjaidea on testata ohjelman osia sekä napata virheitä ja muita ongelmia ennen kuin ne unohtuvat kokonaan seuraavassa ohjelman iteraatiossa (10). Jatkuva integraatio on hyvin työläs ja aikaa vievä prosessi, yksikkö- sekä testausautomaatio vie aikaa ohjelmistokehittäjältä ohjelman kehittämiseen (3). Tämä johtaa suuriin kuluihin ja minimalistisiin ominaisuuksiin tuotteessa. Integraatitesteissä ajetaan kaikki projektiin liittyvät yksikkötestit, jotta saadaan varmistettua funktioiden ja luokkien toiminnallisuus, eikä olla työntämättä tuotantoon virheellistä koodia (9).

### 3.7 Manuaalinen testaus

Manuaalinen testaus on prosessi, jolla etsitään virheitä ohjelmasta (kuva 6). Toteutuksessa yritys palkkaa ammattilaistietokoneohjelmisto testaajia kokeilemaan tuotteen toiminnallisuutta. Manuaalitestaaja käy läpi ohjelmaa asiakkaan sekä käyttäjän näkökulmasta. Lisäksi manuaalitestaaja ajaa yksikkötestejä virheiden löytämistä varten, eikä

käytä testauksen automatisointi työkaluja. Testaaja suunnittelee dokumentin, jossa kuvataan tarkasti lähestymistapaa elementin testaamisessa. (11.)



**Kuva 6** Manuaalitestauksen prosessisykli (11)

Manuaalitestauksen prosessisyklissä on kuusi askelta, jonka toteutuksen jälkeen toistetaan askeleet. Tämä on yrityskohtainen prosessi, eikä vastaa jokaista manuaalitestaus yrityksen lähestymistapaa ohjelman testauksessa.

1. Vaatimusmäärittely (requirement analysis), jossa pyritään määrittämään tarpeet ohjelmatoiminnallisuudelle (11). Nämä vaatimukset ovat tyypillisesti dokumentoitu, niitä on mitattu, testattu, ovat helposti löydettävissä, ottavat huomioon liiketoiminnan tarpeet ja ovat hyvin tarkasti määritetty.
2. Testaus suunnitelma (test plan creation) on dokumentti, jossa pyritään hahmottelemaan toimintasuunnitelma (12). Siinä käydään läpi, mitä tavoitteita on saavutettava, testin pituus, testin arvo, sekä paljon aikaa ja resursseja sen luomiseen tullaan tarvitsemaan.



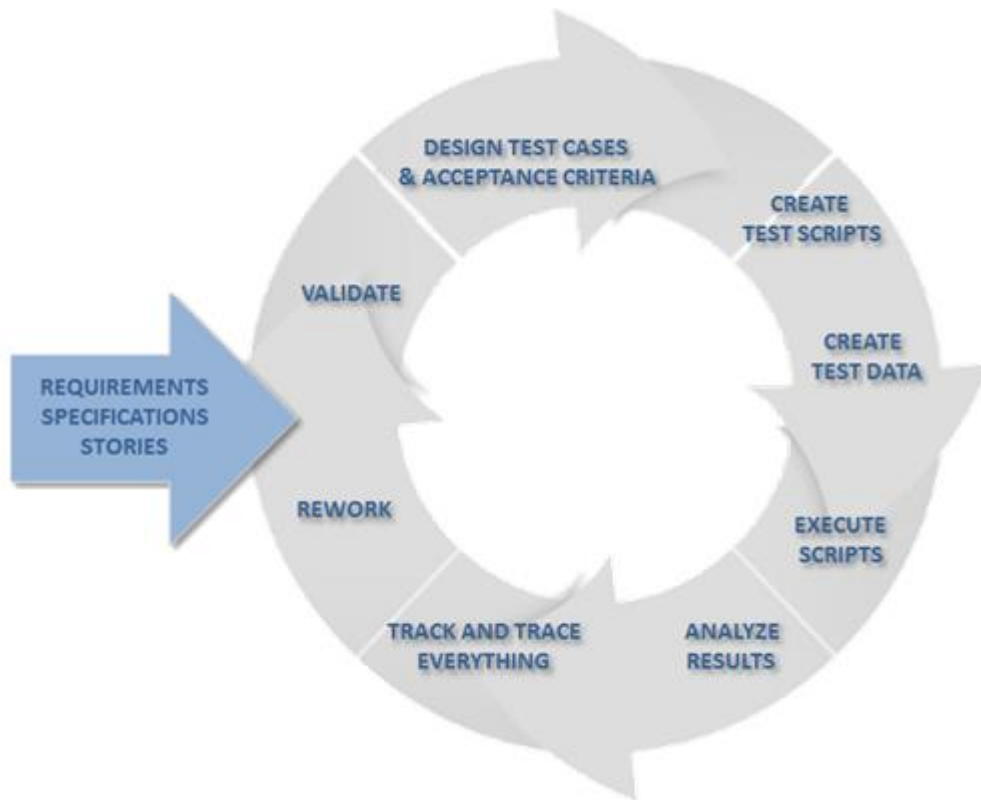
3. Testitapaus (test case creation & execution) on alustavasti luotava (13). Testitapaus on dokumentti, jossa listataan erilaisia toimintoja mitä pitää testata sovelluksessa, sen tarkoituksena on selkeyttää ohjelmistokehittäjälle, mitä komponenttia halutaan testata ja miten.
4. Kun testitapaus on luotu sitä voi lähteä suorittamaan, jotta voidaan varmistaa toteutuksen toiminnallisuus virheettömäksi (14).
5. Virheen kirjaus (defect logging) on prosessi, jossa nauhoitetaan testauksesta tullut palaute virheen korjausta varten (11).
6. Manuaalitestauksen viimeinen vaihe on tarkistaa (defect fix & re-verification), että onko virhe korjattu (11). Jonka jälkeen se kirjataan selvitettyksi vai esiintyykö virhe vieläkin ohjelmassa. Sen jälkeen virhe kirjataan selvittämättömäksi.

Yritykset pyrkivät tyydyttämään asiakasta. Siksi hyvämaineiset yritykset haluavat luoda hyvän vaikutteen asiakkaille, jotta yrityksestä jää positiivinen kuva. Manuaalitestauksen avulla syntyy tuotteelle laatua. Tuotteen laadukkuus antaa asiakkaalle kuvan luotettavasta ja ammattimaisesta yrityksestä.

### 3.8 Testausautomatisointi

Testausautomatisointi on prosessi, jossa ajetaan ohjelman testit skriptikäskyjen avulla (15). Testikäskyt JSON-paketissa vaativat tietokoneen ajamaan testejä kansiosista. Kansion sisällä löytyy jokaiselle oleelliselle komponentille olemassa oleva testitiedosto, joka testaa elementtejä, nämä tiedostot ovat eritelty komponenttien mukaan. On otettava huomioon myös skriptit, joilla voi validoida dataa, jos aikomuksena on testata tietokantaa (15).

Ennen kuin harkitsee testausautomatisointia, on analysoitava omaa projektia ja mietittävä jos se on tarpeen (16). Testausautomatisointi on kallista ja se myös hidastaa projektin valmistumista (6). Ympäristön on oltava myös stabiili, eikä saa tulla liikaa virheellistä esiintymää testipalautteissa (16). Kuvassa 8 käydään läpi testausautomatisointi prosessisykliä.



**Kuva 7 Testausautomatisointi prosessisykli (17)**

Testausautomatisointi prosessisyklissä on kahdeksan askelta, kun suoritetaan testausmäärittelytavoitteita. Toteutuksen jälkeen toistetaan askeleet. Tämä on yrityskohtainen prosessi, eikä vastaa jokaista testausautomatisointi lähestymistapaa ohjelman testauksessa.

1. Validoinnissa (validate) halutaan tarkistaa, että toteutus toimii oikein. Sitä varten on luotava menetelmä, jonka avulla voi tarkistaa muutosten vaikutus sovellukseen (17). Testauksessa validoinnilla tarkoitetaan, että ohjelma testaa tuotteen toiminnallisuutta.
2. Suunnittelu sovelluksen testitapauksille sekä hyväksymiskriteereille (design test). Testitapauksien avulla ohjelmistokehittäjä luo kohde komponenttiin testejä. Hyväksymiskriteerit testeille luodaan laadunvarmistussyistä. Ohjelmistokehittäjä ei halua luoda epävakaita testejä, jotka hidastavat kehitysprosessia. Hyväksymiskriteerien avulla pystytään seuloa läpi huonoista toteutuksista. (17.)
3. Testausskriptien luonti (create test), tyypillisesti komentoriville syötetään käsky, jonka jälkeen skripti aktivoituu ja testausprosessi alkaa (17). Yritykset haluavat

ohjelmistokehittäjien viettävän vähemmän aikaa testikomponenttien kirjoittamiseen (36). Ajatuksena on kuormata avainsanoja skripteillä (36), jonka jälkeen testaajat käyttävät avainsanoja testispeksien luontiin (36).

4. Testi datan luonti (create testdata). Testaamisessa vaaditaan testidataa, jolla testata elementtien validointia ja toimintoja. Oikean datan käyttö on liian iso riski, joten on luotava omaa testaus data. Manuaalisesti datan luonti voi olla hyvin työläs prosessi. Markkinoilla on nykyään työkaluja, jotka voivat luoda tietyn tyyppistä dataa, riippuen sovelluksen tarpeisiin. (17.)
5. Skriptien ajo (execute). Testien automatisointi työkalu tulee olemaan koko toteutuksen keskus. Työkalun valitseminen tulee vasta sen jälkeen, kun työympäristö, johon aletaan rakentamaan projektia, on päätetty. (17.)
6. Tulosten analysointi (analyze), tuloksia tulee suoraan siitä, kuinka moni testeistä onnistui sekä epäonnistui (17). Lisäksi on tehtävä tarkat määrittelyt sitä varten, mikä lasketaan onnistumiseksi ja mikä on sitten epäonnistuminen.
7. Testi tulosten seuranta (track). Työkalun on osattava muutakin kuin pelkästään osoittaa testikattavuutta ohjelmassa. Testikattavuuden lisäksi on oltava myös ilmoitus siitä mitä testiltä oli odotettu, mikä oli tulos, mikä testi epäonnistui ja missä testi sijaitsee. Nämä tarpeelliset ja hyvät käytännöt tulevat nopeuttamaan projektin kehitystä. (17.)
8. Kehitys prosessin parannus tulosten pohjalta (network) (17). Katsotaan mitä tuli opittua ja parannetaan nykyistä toteutusta.

#### **4 Protractor-testausautomatisointikehikko**

Protractor on Node.JS:n sisään rakennettu testauskehikko. Sitä käytetään pääosin End-to-End-testausautomatisointityökaluna, joskus sitä sekoitetaan End-to-End-tietokoneohjelmistoksi. Yksikkötestit toimivat ohjelmoinnin näkökulmasta, eli luokalta odotetaan jotain tiettyä toteutusta. End-to-End-testeissä testataan käyttäjän näkökulmasta. (19.)

Testien luominen Protractorissa on helppoa, mutta testien ylläpito ja laadukkuus on hankala säilyttää. Valitettavasti on vältettävä tapahtumien (event) ja lupauksien (promise) testaamista, sillä Protractor ei kykene aina löytämään näitä komponentteja luotettavasti. On suositeltava ottaa käyttöön staattista koodianalyysityökalua, joka sitten merkitsee eihaluuttua koodityyliä virheenä ohjelmassa. Kuten eslint pakottaa tietyn tyyppisiä käytäntöjä koodissa, strict mode (ohjelma asetus) suositellaan ottamaan käyttöön, koska se poistaa tyypilliset virheet, joita ei kerralla aina onnistu huomamaan. Versionhallintaohjelma testaa säiliötä vetopyynnöllä (pull request), mikä antaa muiden tiedostaa muutoksia, tehtyyn säiliöön. Samalla se puskee kaikki tallennetut muutokset omassa työhaarrassa. (19.)

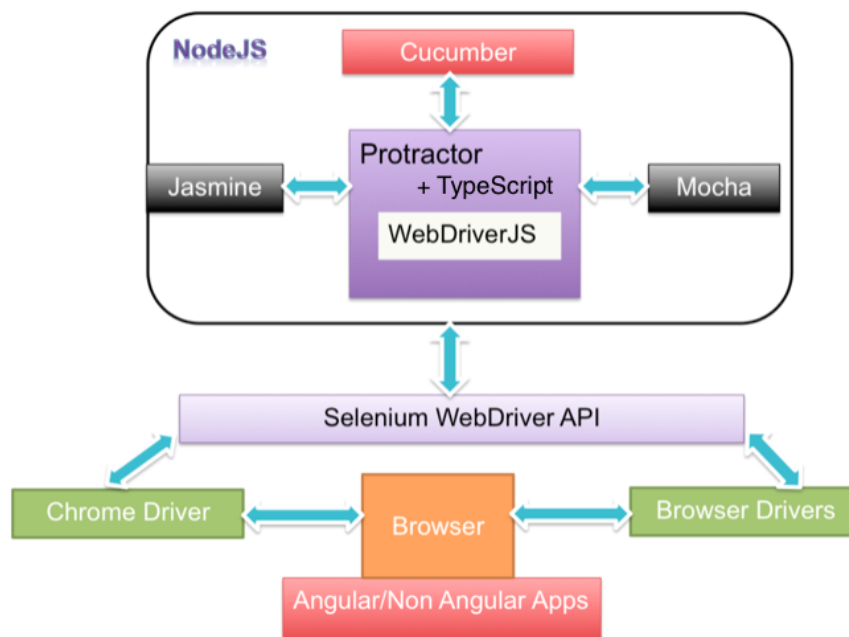
Koska jokaisessa testissä on kovakoodattua dataa, sitä suositellaan säilyttämään konfiguraatitiedostossa, jossa sitä voidaan muokata ja tarkistaa. Kirjastossa nimetään elementit, joita on testattava. Se nopeuttaa testien luontia ja voidaan käyttää samaa funktiota uudestaan muualla koodissa. Protractor suosii XPath-selektorien käyttämistä (`$("#img")`), niillä pystyy valikoimaan elementtejä selaimesta. Voi myös toki käyttää CSS:ään löytääkseen elementit sekä merkitä elementit ID:llä. XPathin idea on matkia valinnan tekoa käyttäjän näkökulmasta. (19.)

Jotta tuote pysyisi laadukkaana, on pystyttävä odottamaan ohjelman alustusta selaimelle. Protractor tarjoaa `protractor.ExpectedCondition` luokan, jolla voi käsitellä Angularin pitkää lataus aikaa sovellukseen. Nämä edellytykset ehkäisevät turhia virhetilanteita. (19.)

Protractoriin voi asentaa raportointijärjestelmämoduulin. Moduuli tulostaa html-sivulle onnistuneet sekä epäonnistuneet testit. Lisäksi se ottaa kuvakaappauksen virhetilanteesta, jotta virheen löytäminen onnistuisi nopeasti. Tämä moduuli ei aina onnistu nappaamaan virhetilanteita kuvakaappauksilla. Tällöin syntyy tilanteita, jossa näkyy vain tyhjä sivu. (19.)

Kuvassa 9 näkyy, kuinka Protractor integroi tarpeellisia työkaluja järjestelmäänsä.

## Protractor-Cucumber Test Automation Framework



**Kuva 8 Malli miten rakentaa Protractorille toimintaympäristö (20)**

NodeJS, Selenium, WebDriverJS, Jasmine, Cucumber ja Mocha tekevät End-to-End-testaamisesta helposti skaalautuvat ohjelmistolle (20). NodeJS:llä luodaan Front-End-palvelimia, jolla sitten voi toteuttaa käyttöliittymäsovellusta JavaScriptillä. Selenium etsii muutoksia selaimen ympäristössä. Protractor tarvitsee WebDriverJS:n, joka mahdollistaa Selenium WebDriver API:n käytön, automatisoidakseen selainajureita sekä ajaa Jasmine-yksikkötestejä selaimelta. Cucumber on testaussovellus, jolla testataan sovelluksia. Mocha on kehikko, jolla voi suorittaa asynkronisia testejä sekä tulostaa testikattavuusraportteja. Angular ei ole pakko käyttää ainoana sovellusalustana (20).

### 4.1 Asennus

Alustavasti on päätettävä, haluaako käyttää Node.JS-palvelinta kehykkona projektille. Sen jälkeen on asennettava Node.JS. Jos luodaan projekti, tarvitaan projektille polku. Tyypillisesti projekti sijaitsee suoraan tietokoneistojuuren alla.

Kun Node.JS on asennettu, syötetään komentoriville `npm install -g protractor`. Asentajan on oltava projektiin hakemistossa, jonne halutaan luoda testitapaukset (21). Tämän jälkeen on luotava testiaustiedostosi rinnalle `conf`-tiedosto. Sinne on vähintään kirjoitettava `exporters.config` (21). Sen sisällä on määritteitä sekä komponenttikansioita, jossa ajetaan testitapauksia (21).

## 4.2 Protractorin käyttökokemus

Protractorin ajaminen on suhteellisen helppoa. Protractor.JSON-tiedostossa on skripti nimeltään "npm e2e test". Komento ajaa kaikki testit, jotka on liitetty sovellukseen. Ne sijaitsevat `specs`-kansiossa. Testit ajetaan tiedostojärjestyksessä, oletuksena testit ajetaan `headless`-tilassa. Se tarkoittaa, että ikkuna ei avaudu ja testit ajetaan ilman, että käyttäjä näkee, mitä tapahtuu. Kun halutaan nähdä, mitä Protractor tekee näytöllä, `headless`-käsky poistetaan tiedostosta. Silloin päästään ajamaan skriptiltä End-to-End-testi komentoa, ja nähdään, mihin se jää jumiin näytössä. Protractorin asetustiedostossa voi määritellä, millä sivulla ajetaan testit. Siellä voi myös asettaa sivujen lukumäärän ja sen, millä kaikilla sivuilla ajetaan testit. Tiedostossa voi myös muokata, missä osoitteessa ajaa testit. Yleensä testit ajetaan `Debug`-tilassa (tutkintaohjelma jolla voidaan etsiä ohjelmistovirheitä ohjelmassa) omalla koneella osoitteessa `localhost:3000`.

Virheviesti ilmestyy koonnissa näkyville projektin kehitysryhmän Team Foundation Serverille. Team Foundation Server on Microsoftin omistama SaaS, jossa voi tehdä tilauksissa muutoksia ja ohjata tiimiä, ei saada virhettä heti napattua. Yleensä jos on toistuva virhe, viesti on tarpeeksi selkeä, jotta tiedetään, mikä hajoaa ja miksi. Tarkennettua virheen tulostusta on vaikea lukea. Vaikkapa alasvetovalikon hajoaminen tietyssä sivussa on vaikea lukea ilman uudelleenajoa. Yleensä testit ajetaan omalta koneelta uusiksi testipalvelimella, mutta vain jos kukaan muu ei ole käyttämässä sitä sillä hetkellä. Jos se on käytössä, asetuksissa voidaan vaihtaa osoitteeseen `localhost`. Mutta vain jos `localhost`-palvelin on pystytetty. On hyvä pitää haara ajantasalla palvelinversion kanssa, joten yleensä kannattaa päivittää master-haara itselleen.

Virheentunnistus on heikkoa. Kun palvelimella ajettu koonti epäonnistuu, virheen palaute tulee olemaan hyvin epäselvää. Hyvin yleisesti palautteen sekaan on liitetty virheeksi jotain ydinongelmia Angularissa. Nämä ilmoitukset eivät auta virheen korjaamisessa.

Eräs tapa kiertää ongelma on tehdä useita kuvausosiota testitapauksiin. Kuten ”Käyttäjätiletestit” ja sen alla yksi testiosio ”Markon tila muuttui”. Tämä johtaa melko nopeasti hyvin työlääseen prosessiin ja saa aikaiseksi tuhat riviä koodia, mutta vain neljäsosa sovelluksesta on testattu. On tehtävä tarkat kuvaukset, missä testit epäonnistuvat, mutta ei lisätä jokaiselle elementin kosketukselle kuvausta. Pyritään enemmän osoittamaan, missä virhe tapahtuu ja sitten testaaja ajaa vain yhden osion omalla koneella uusiksi.

## 5 Muut automaattitestaushkeikot

Protractor ei ole ainoa automaattitestaushkeikko, jota voi käyttää Angularin kanssa. Protractorilla on ongelmia havaita komponentteja, jotka ilmentyvät selaimelle tiettyjen kriteereiden jälkeen. Se myös ei aina kykene ajamaan haluttua toimintaa, joka johtaa testien epäonnistumisiin. Omituisempaa on, kuinka hidas Protractor on, kun se joutuu odottamaan Angularin latautumista selaimelle.

### 5.1 TestCafe

TestCafe toimii suosituissa käyttöjärjestelmissä (Windows, MacOS ja Linux). Siinä on tuki kaikille selainmuodoille. Asennus yhtä helppoa kuin Protractorissa. Kun Node.JS on asennettu, syötetään komentoriviin `npm install -g testcafe`. TestCafe on ilmainen, ja se käyttää MIT-lisenssejä. Liitännät tarjoavat monipuolisia työkaluja sekä kykyä ajamaan testit ohjelmointiympäristöstä. Liitännöillä voi myös luoda oman GitHub. GitHub tarjoaa versiohallintaa ohjelmistokehitysprojekteille yhteisön. (22.)

### 5.2 Cypress.io

Cypress.io on avoimen lähdekoodin testauskehikko. Cypress ei vaadi riippuvuuksia tai lisäasennuksia ohjelmaan (23). Koodikieltä ei myöskään tarvitse muuttaa (23). Sen avulla voi nähdä kaikki testit selaimen rinnalla, jotain mitä Protractor ei tarjoa, sillä testit ajetaan komentoriviltä ja testit myös tulostuvat komentoriville. Cypress antaa mahdollisuuden etsiä virheitä käymällä testausvaiheita läpi. Se myös osoittaa vaiheen selaimessa (23). Sillä voidaan myös nähdä mitä tapahtuu, kun valitaan toinen komento debuggerissa. Asennuksessa vaaditaan projektin polku sekä komento `npm install cypress --save-dev` komentoriville (23).

### 5.3 Nightwatch.JS

Nightwatch.JS on Node.JS pohjalta luotu kehikko (24). Se käyttää W3C WebDriver API:a toteuttaakseen komennot sekä julkaisut testeistä (24). Yksinkertainen lauseoppi, jolla on mahdollista kirjoittaa testejä käyttäen Javascript-kieltä (24). CSS:n tai Xpath:n avulla voi osoittaa haluttuun elementtiin selaimessa (24). Nightwatch.JS:illa on Grunt-tukea (24). Grunt hallinnoi Selenium-palvelimella eristyksessä, jota sitten voi myös ottaa pois käytöstä. Integraatiotestit voi lisätä koontiin ja ajaa testit tuotannossa (24). Sisältää pilvitukea sekä saa ohjelman myös laajennettua (24). Asennus vaatii mennä Nightwatch.JS-sivuille ja sitten suorittaa asennuspaketti.

## 6 Muita testien automatisoinnin työvälineohjelmia

Luvussa käydään läpi yleisellä tasolla oleellista tietoa testausautomatisointiyrityksistä, sekä heidän menetelmiä hallinnoida laatua projektissa. Ne ovat tapoja, joilla voi tehdä End-to-End-testeistä kestäviä ja mitä mallia kannattaa seurata, että voi välttää suuria hikkoja projektin kehityksessä. Miten käyttää End-to-End-testausautomatisointia mahdollisimman tehokkaasti ilman viivästyksiä projektin kehittymiselle.

### 6.1 Näkemys testausautomatisoinnista

Koska testausautomaatio on paras tapa lisätä tehokkuutta laadun varmistamiselle tuotteessa, se myös parantaa työtehokkuutta. Manuaalista testausta pitää tehdä vähemmän, testausautomaatio on työkalu, joilla testaus helpottuu. Kun testitapaus luodaan, testejä voidaan toistaa sekä laajentaa, niitä voidaan parantaa ja monimutkaistaa tavoilla, joita manuaalitestaus ei saavuttaisi millään. (25.)

Siitä asti, kun testausautomaatio luotiin, sitä on pidetty hyvin tärkeänä ja isona osana ohjelmiston valmistukselle (25).

Valitettavasti testausautomatisoinnin toteutus saattaa olla haastavaa nykyiseen API-järjestelmään (ei koske Node.JS). Testausautomatisointi säästää rahaa ja aikaa. Jokaisen kehitysjakson jälkeen kerätään virheet, joita ohjelmistokehittäjän on korjattava ja hyvin usein sama bugi toistuu yhden jakson jälkeen. Aina kun tehdään muutosta Front-End-



puolen koodiin, on ajettava testit uudelleen, jotta varmistetaan, ettei muutos hajottanut mitään. (25.)

Kukaan ei voi testata manuaalisesti jokaisen muutoksen jälkeen koko sovellusta varmistukseen, että kaikki elementit toimivat oikein, eli tehdään regressiotestausta. Testitapauksen luomisen jälkeen testejä voidaan ajaa sekä muokata loputtomiin ilman lisäkustannusta ja ajanhukkaa. (25.)

Testausautomatisointi lisää kattavuutta, testien laatu paranee ja testit muuttuvat luotettavimmiksi. Testejä voidaan ajaa monissa eri selaimissa erilaisilla asetuksilla samaan aikaan palvelimessa. Testausautomaatio kykenee näkemään sovelluksen sisälle sekä osaa määrittellä tuotteen toimivuuden. Automaattitestit kykenevät ajamaan tuhansia erilaisia mutkikkaita testejä jokaisen ajon jälkeen. (25.)

Testaaminen parantaa ohjelman laatua. Manuaalitestaajat tekevät usein virheitä testaamisen aikana, eivätkä aina kykene seuraamaan samaa yhdistelmää tuottaakseen virheen uudelleen. Automaattitestauksessa voi toistaa testiä useamman kerran. Se myös kykenee nauhoittamaan tulokset. Yksinkertainen helppokäyttöinen testausautomatisointiohjelma vapauttaa ohjelmistokehittäjät takaisin ohjelmistokehityksen pariin ja tekee manuaalitestaajista ohjelmistotestaajiksi. (25.)

Testausten automatisointi kykenee sellaiseen, mihin normaali manuaalitestaus ei pysty. Suurimmatkaan laadunvarmistusyrietykset eivät kykene siihen, mihin tietokone kykenee. Testausautomatisointi pystyy ajamaan tuhansilla virtuaalikäyttäjillä laadunvarmistustestejä ongelmitta. (25.)

Laadunparannustyökalu kuten TSLint on joustava staattinen koodin tarkastustyökalu, joka tarkistaa koodin laadun. TYPESCRIPTLINT on JSON-tiedostopäätteinen ja sen sijainti on kiinnitetty pääprojektiin. Tiedoston sisältä löytyy staattisia määräyksiä, joita työkalu käy läpi ja tarkistaa ohjelmistokehittäjän kirjoittamaa TypeScript-koodia. Jos ohjelmistokehittäjä on lisännyt virheellistä koodia, testaus työkalu löytää sen ja ilmoittaa ohjelmistokehittäjän virheestä. Tämä säästää ohjelmistokehittäjältä turhaantumista ja luo itsevarmuutta omaan työhön. Jos tietokone ei löytänyt mitään väärää, niin virheitä ei myöskään löydy. Niin laatutiimin ja kehitystiimin työmoraaali paranee, yksikön taito paranee ja huolet vähenevät, koska on olemassa joku verkko, joka nappaa virheet ja ilmoittaa ne. (25.)

## 6.2 Selenium

Selenium tekee HTML -koodista lukukelpoisen, jotta End-to-End-testejä voidaan suorittaa selaimessa. Ilman Seleniumia kehikko ei kykene lukemaan selaimen elementtejä testaamista varten. Selenium pyörii palvelimessa, jonka sisällä ajetaan End-to-End-testaus automaatiota. Sen mukana tulee testaus-työkaluja kehikoita sekä API:ja. Selenium tarjoa selainajurit sekä ohjelmointiympäristön ladattavaksi sivustollaan (26).

## 6.3 Katalon Studio

Katalon Studio on automaatiotestaustyökalu mobiili- ja verkkoympäristöille. Se rakennettiin Seleniumin ja Appiumin runkorakenteesta. Katalon Studion avulla kokematon ohjelmistokehittäjä pääsee nopeasti vauhtiin, koska runkoon on sisäänrakennettu satoja avainsanoja. Sillä on myös työkaluja, jotka auttavat skriptauksessa, ja ohjelma kykenee nauhoittamaan testiajoja. Runko toimii mainiosti qTest, JIRA ja GIT:in ohjelmien kanssa. Se myös tukee BDD-syntakseja. Ohjelma käyttää Apache Groovy-skriptauskieltä, testausajoissa käytetään JRubt ja Jythonia. Katalon Studio tukee Java-kirjastoja sekä muokattuja kirjastoja. Katalon Studiolla pystytään tekemään End-to-End-testejä web-, mobiili- ja ohjelmointiympäristöissä. (27.)

## 6.4 Watir

Watir on avoin rajapinta Ruby-kirjastolle, jolla voi ajaa End-to-End-testausta selaimesta (28). Ohjelma pyrkii käyttäytymään kuin ihminen selaimessa (28). Ohjelma testaa linkkejä, täyttää lomakkeita ja validoi tekstejä (28). Watir tarvitsee Selenium-runkorakennetta sekä Rackiä lisätoiminnallisuuksiin (28).

## 6.5 Rational Functional Tester

IBM Rational Functional Tester kaappaa kuvan epäonnistuneesta testistä. Tämän pitäisi helpottaa ongelman löytämistä (29). Valitettavasti kuvan kaappaus näyttää vain tilanteen lopun. Se ei näytä miten tilanteeseen päästiin. Kuvankaappaus on nopea tapa katsoa, missä vika lienee, mutta tarkempaa tietoa saa vain testiskriptin ajossa (29).

ScriptAssure yhdistää samankaltaiset algoritmit löytääkseen oliota, jotka toimivat avainsanoilla. Tämä edesauttaa manuaalitestaaajia ajamaan testiskriptejä automaattisesti (29).

IBM Rational Functional Tester voi asettaa parametreille tiettyä dataa ja tarkistaa datan sen lisäämisen jälkeen (29). Tarkoituksena on tarkistaa, jos syötetty data tulostuu oikein elementille.

Se toimii sekä Javassa että Visual Basic.Netissä, mikä tarkoittaa, että ohjelmointiympäristönä toimii Eclipse ja Visual Studio (29). Molempiin on sisäänrakennettu skriptiä avustavia työkaluja, jotka mahdollistavat testausautomaation (29).

## 6.6 Robot Framework

Robot Framework on tavanomainen testausautomaatorunko, joka keskittyy hyväksymistesteihin sekä hyväksymistestien testivetoiseen kehitykseen. Se käyttää avainsanates-tausmetodia. Se helpottaa manuaalitestaaajien työntekoa, koska se on helppolukuista eikä vaadi ohjelmointiosaamista. Valitettavasti jokaisella avainsanalla on oma funktio tai toimintatapa ja avainsanan muokkaaminen vaati ohjelmointitaitoja. Kaikki testitapaukset sijaitsevat taulukkomuodossa, mikä helpottaa testien lukemisen. Yhteensopivat kirjastot ovat Python sekä Java. Käyttäjät kykenevät määrittelemään saman syntaksin pohjalta uusia avainsanoja, mutta ne pitää ohjelmoida.

Robot Framework on riippumaton käyttöjärjestelmästä sekä sovelluksesta. Python, Jython ja IronPython kykenevät ajamaan ydinrunkoa.

## 6.7 Eggplant

Eggplant-tekoäly käyttää analyyttisiä menetelmiä ratkaistakseen ongelmia. Se pyrkii en-nustamaan, missä syntyy laadunvarmistuksen vastaisia ongelmia parantaakseen testausyksiköitä löytämään ja korjaamaan virheitä nopeasti. (31.)

Kaikki kattava testaus API:ssa on lähes mahdotonta, ohjelmointitiimi on hyvin pieni eikä se kykene matkimaan tuhansien käyttäjien päätöksentekoa. Jokaisessa sovelluksessa tulee aina olemaan loogisia virheitä tai bugeja. Uuden Eggplant-tekoälyn avulla ohjelmistokehittäjät löytävät virheet ja bugit nopeasti ohjelmassa. Tekoälyn avulla ei enää

tarvitse luoda testitapauksia käyttäjätarinoiden pohjalta, vaan voi keskittää testien luonnin todellisissa ongelma alueissa. (31.)

Tekoäly käyttää neuroverkkoa hyödykseen ja luo testejä sekä keskittää käyttäjätarinoiden pohjalta testitapausten luontia (31).

## 6.8 Tricentis: API-testaaminen ja ylläpito End-to-End-testausautomaatiossa

Tricentis opastaa luomaan osia, joita voi hyödyntää End-to-End-integraation testaamisessa. Päivitykset sujuvat helpommin, koska nykypäivän API:t käyttävät mallipohjaista testausautomatisointia. (32.)

Tricentis päivittää ohjelmaan tehdyt muutokset automaattisesti. Elementtien muutokset voi päivittää ja ohjelma tekee tarpeelliset korjaukset taustalla automaattisesti. Periaate on sama kuin jatkuvan koonnin ohjelmointiympäristössä. (32.)

Tärkeimmät ominaisuudet ohjelmassa voi turvata asettamalla ne tärkeysjärjestykseen. Testit ilmoittavat haittatekijät ongelmien syntymisille ja ehdottavat toimintatapoja vaarojen minimoinnille. Lopputavoitteena on saada mahdollisimman laaja riskikattavuus. (32.)

Työkalujen yhdistämisessä parannetaan yleisellä tasolla testien laatua sekä pyritään antamaan lisää joustoa projektin kehitykselle. Tarkoituksena on vähentää riskitekijöitä End-to-End testeissä. Oleellista on kuitenkin varmistaa, että työkalut ovat yhteensopivia, eivätkä aiheuta lisää virheitä ohjelmalle. Tricentis ratkaisuna on yhdistää käyttöliittymät niin, että ohjelma on yhteensopiva mobiili-, websovellusten sekä suurtietokoneiden kanssa. Ohjelma tulostaa yhdistettyä testidataa työnjohdosta niin, että se toimii yhdessä parhaimpien DevOps-ohjelmointityökalujen kanssa. (33.)

## 6.9 Kattava automaatiotesti -työkalu

Automaatiotestityökalu on työympäristötestien nopeutustyökalu. Sillä voi suorittaa web- ja mobiilitestejä applikaatiolle. Useimmat automaatiotestityökalut yleensä toimivat vain web- ja mobiiliympäristöissä. Automaattitestejä voi ajaa Windowsin työpöydissä joko etänä tai paikan päällä. IOS- ja Android-mobiililaitteet pystyvät ajamaan sekä emulaattoreissa että simulaattoreissa. (34.)

Rinnakkaistestaaminen on myös mahdollista ja testaaminen onnistuu seuraavilla selaimilla: Chrome, Firefox, Safari ja MS Edge (34).

Ohjelman päämääränä on lisätä laatua ja vähentää toistuvien ongelmien korjaamista (34).

## 6.10 Qentinel

Qentinel pyrkii tekemään manuaalitestaaajasta End-to-End-testaajan. Tarkoitus on siirtää End-to-End-testaaminen ohjelmistokehittäjästä manuaalitestaaajan työksi. Testaus tulisi toteutumaan koordinaatiston ja avainsanakoodin pohjalta.

Manuaali testaajat pääsevät luomaan testitapauksia eikä enää tarvitse palkata ohjelmistokehittäjän tekemään sitä. Avainsana on olio, jota pitää toteuttaa oikein. Olion luontiin ja sen käyttäytymiseen vaaditaan koodaustaitoja. Testausohjelma käyttää HTML-rinnastusta löytääkseen elementtejä, joita voi muokata. Ongelmana on yleensä elementit, jotka ilmestyvät jälkepäin. Esimerkkinä ovat, ponnahdusikkunat, jotka syntyvät API:ssa vasta toisen elementin kanssakäymisen jälkeen. Ne ilmestyvät HTML:ssä pohjimpana, mikä vaatii toteutuskikkailua. (36.)

Environment as a Service. Ideana on luoda palvelu, jota voi myydä ympäristönä useammalle ohjelmistokehitysyritykselle (36). Se helpottaa testien uudelleenkäytössä. Todella kätevää, jos on useampi sisaryhtiö, ei tarvitse lisensoida kaikki erikseen. Jos kilpaileva yritys käyttää samaa ympäristöä, kilpailukyky heikkenee. Jos useammalla yrityksellä on sama ympäristö, voi tämä johtaa tietoturvaongelmiin sekä hintaviin oikeuskäynteihin. Yrityksen tuottamat ohjelmat ovat liian erilaisia toteutusympäristöä varten, hyödyntääkseen ratkaisua.

## 7 Analysointia

Alustavasti on harkittava End-to-End-testausautomaation budjetti, ennen kuin ryhtyy implementoimaan sitä projektiin. Kun testausautomaatio päätetään lisätä projektiin, sitä varten kannattaa hankkia tarpeenomaista henkilöstöä, jotka tulevat käyttämään ohjelmaa. Kyseessä on todella työläs lisäys, joka vaatii aktiivista tiimiä toimiakseen oikein.

Lisäksi suositellaan myös mietittävä ohjelman monimutkaisuutta. Testausautomatisointia ei ole välttämätöntä lisätä projektiin. Se eroaa muista testausmenetelmistä käyttäjämäisellä testaustavallaan. Yhtä hyvin voi toteuttaa projektin osien testaamisen Mock-palvelimen avulla. Saadaan nopeamman ja halvemmän tavan toteuttaa osien laadun tarkistus.

End-to-End-testausautomaatiosta on enemmän iloa manuaalitestaaajille kuin muille projektin jäsenille, sillä manuaalitestaaajat tarvitsevat regressiotestausta varten työkalua, joka nopeuttaa toistuvaa testaamista. End-to-End-testauksen automatisoinnin heikkoutena pidetään ohjelmistokehittäjän aktiivista roolia testien luonnissa. Ylläpitoon sekä testien luontiin kuluu paljon aikaa ja sitä aikaa voisi käyttää tuotteen kehitykseen.

Tällä hetkellä, jos ottaa testausautomatisoinnin projektiin, kannattaa tehdä vähän molempia, mutta pyrkiä välttämään liiallista End-to-End-testausta. Tiimin on pyrittävä pysymään pyramidimallissa, koska se olisi vähiten resursseja vievä käytäntö. On myös suositeltava pitää End-to-End-testit erillään koontitesteistä. Tiimin on myös osattava rajata End-to-End-testaamista projektissa, jotta ohjelman kehitys pysyisi ajan tasalla.

Testaaminen ja laadun varmistus tulee kalliiksi yritykselle, mutta samaan aikaan se maksaa itsensä takaisin laadun varmistuksen kautta. Valitettavasti End-to-End-testausautomaation oikeaa hyötyä ei voida todistaa käytännössä. Virheiden löydös ei vaikuta asiakkaaseen millään tavalla. Se mikä vaikuttaa, on, kun asiakas itse löytää virheen tai löytää jotain ominaisuutta ohjelmassa, josta asiakas ei pidä.

End-to-End-testauksen automatisoinnin olemassaolo yritysmarkkinoilla on ymmärrettävä, ohjelmistokehittäjät tekevät virheitä tai tuovat haitallisia ominaisuuksia ohjelmaan. Testauksen automatisointi pienellä tapaa paikkaa virheet ja osoittaa yrityksen pyrkineen mahdollisimman laadukkaan tuotteen kehitykseen, mutta se ei ole mikään automaaginen ratkaisu vialliselle ohjelmalle.

Jos harkitsee End-to-End-testausautomatisoinnin käyttöä projektissa, on olemassa kaksi tarpeeksi erilaista toteutustapaa, jota kannattaa harkita ennen kuin lähtee työstämään toteutusta.

Tapa1 on avainsanoilla ohjattua testaamista. Ajatuksena on antaa manuaalitestaaajan tehdä komponenttitestejä. Manuaalitestaaaja yhdistää avainsanoja toisiinsa, jotka sitten

aktivoivat koodifunktion. Nämä funktiot sisältävät käskyjä. Funktion sisällä on nimiä sekä koordinaatistoja, jolla voi hakea elementtejä, mitä halutaan testata sovelluksessa. Yhdistelmät kuten "painike" ja "paina" testaa painikkeen painalluksen. Jos painiketta ei voi painaa tai sitä ei löydy testi palauttaa epäonnistumisen. Avainsana-"painike" aktivoi funktion, jossa etsitään painikkeita koordinaatistolla. Koordinaatisto on metodi, joka havaitsee painikkeen tietämällä, missä järjestyksessä ohjelma luo koodia sovellukselle. Elementteihin ei enää tarvitse lisätä ylimääräisiä tunnisteita, joilla löytää elementti nopeammin. Avainsana "paina" aktivoi funktion, jossa testataan, että painike on olemassa ja sitä voidaan painaa. Kun testi epäonnistuu, se palauttaa takaisin ilmoituksen, missä tarkennetaan virheen alkuperää. End-to-End-testit implementoidaan integraatioprosessiin, testit ajetaan aina, kun tehdään suurta muutosta sovellukseen.

Tavasta 1 löytyy ongelmia itse avainsanoissa. Avainsanat ovat monimutkaisia, mikä tarkoittaa, että niitä on silti ohjelmitava, kun ne hajoavat. Avainsanoja on opiskeltava toiminnallisuuden kannalta ja niiden käyttöön on totuttava. Avainsanat ovat funktiota, jossa on koodia, joten avainsana voi mahdollisesti olla virheellisesti toteutettu, eikä manuaalitestaja huomaa ongelmaa heti. Avainsanoja on ohjelmitava ja ohjelmistokehittäjän on pidettävä muistissa, mikä avainsana tekee mitä, eikä toistaa samoja toiminnallisuuksia, mutta vain eri nimillä. Avainsanojen sisällä voi olla muita avainsanoja, joka tekee korjaustyöstä mutkikkaan. Kun sovellukseen ilmestyy uusia ominaisuuksia, ohjelmistokehittäjän on luotava uusia avainsanoja testaamista varten.

Tapa 2 on ohjelmistokehittäjän käyttö End-to-End-testien kirjoittamiseen. Manuaalitestaja testaa ohjelmaa ja kirjoittaa dokumentteja, missä käydään systemaattisesti läpi, mitä kaikkea on testattava. Ohjelmistokehittäjä kirjoittaa testit, funktiot, jotka tulevat käyttöön toistuvasti lisätään samaan tiedostoon. Tiedostoon viitataan komponenttiin liittyvässä testissä, komponentilla on luotu kaksi tiedostoa End-to-End-testejä varten. Ensimmäisessä on testausajoa varten olevia testejä, toisessa on komponenttitestiä varten kustomoituja funktiota. Testejä ajetaan palvelimessa sovelluksen testiversiossa. Virheen ilmentyessä ohjelmistokehittäjä tiedostaa, jos virhe on testissä tai sovelluksessa. Tarpeen mukaan ohjelmistokehittäjä voi korjata virheen molemmista paikoista. Ohjelmistokehittäjä tuntee koodin ja osaa muokata sekä kehittää testejä eteenpäin.

Myös tavasta 2 löytyy ongelmia. Ohjelmistokehittäjä on kallista palkata tekemään testejä. Ohjelmistokehittäjä voisi olla kehittämässä ohjelmaa eteenpäin. End-to-End-testit hidastavat koontiprosessia sekä ovat herkkiä antamaan virheellistä ilmoitusta. End-to-End-

testejä voi tehdä loputtomasti, mutta sovellusta ei. Tunnisteita on luotava elementeille, jos niitä halutaan testata. Testien tekemiseen kulutetaan paljon aikaa, isot muutokset sovelluksen käyttäytymiselle rikkoo olemassa olevia testejä.

## 8 Yhteenveto

Työ koskee End-to-End-testauksen automatisointia, jolla parannetaan sovelluksen laatua automaattitesteillä. End-to-End-testauskehikoilla pyritään vähentämään virheitä, jotka vaikuttavat asiakkaan kokemukseen ohjelman käytössä. End-to-End-testaus eroaa yksikkötesteistä simuloimalla ihmisen käyttäytymistä ohjelmassa.

Kyseessä on yleiskatsaus End-to-End-testaus automaation olemassaololle. Tässä insinööriyössä ei kuitenkaan ollut tarkoitus kehittää sovellusta, jossa käytetään End-to-End-testausta, vaan tarkastella, mihin sitä käytetään, mitä siitä on hyötyä ja miksi yritykset käyttävät sitä. Työllä on käyttöarvoa, sillä se käy läpi hyödyllistä tietoa End-to-End-testauksesta. End-to-End-testaus automatisoinnissa on paljon hyötyä, mutta siitä on myös haittaa.

Aluksi käytiin läpi End-to-End-testauskonseptia läpi ja selitettiin auki, mikä on sen rooli jatkuvassa integraatiossa. Selitettiin yleisesti, mikä on jatkuva integrointiprosessi projektissa. Käytiin läpi, mikä on Protractor End-to-End-testauskehikko sekä toiminnallisuus. Otetaan huomioon olemassa olevia yrityksiä End-to-End-testausautomatisoinnin ilmesyessä markkinoille, tutkitaan muita testien automatisoinnin työvälineohjelmia. Lopuksi analysoidaan kahden eri firman tapaa toteuttaa testausautomatisointia projekteissaan.

Insinööriyössä käytettiin kirjoituspöytä tutkimusta. Se kohdisti tutkimuksen suunnittelussa, kohdeteknologian valinnassa. Tutkimuksessa päädyttiin tulokseen, missä kohteella ei ole isoa merkitystä alustassaan. Tutkimuksessa analysoitiin aikaisempia kokemuksia End-to-End-testaustavan käytössä sekä henkilökohtaista kokemusta. Tutkimuksessa pyrittiin tuomaan End-to-End-testausautomatisoinnin olemassaolon ohjelmistokehitys alalla sekä tuomaan uusia näkökulmia sen käytölle ja toteutustavalle.

End-to-End-testaamisessa hyötyy päämääräisesti yrityksen organisaatiossa henkilöt, jotka ovat vastuussa tuotteen laadusta sekä toiminnallisuudesta. Yleensä kyseessä on



ohjelman tilaaja, tuotteenomistaja, osakas tai johto. Vaikka ohjelma olisikin avoin lähdekoodi, uuden tavaran lisääminen projektiin tulee yrityksen vastuuhenkilöiltä.

## Lähteet

1. Wacker, Mike. 2015. Just Say No to More End-to-End Tests. Verkkodokumentti.  
<<https://testing.googleblog.com/2015/04/just-say-no-to-more-End-to-End-tests.html>>. Luettu 4.4.2018.
2. Huynh, Phong. 2017. Why End-to-End Testing is Important for Your Team. Verkkodokumentti.  
<<https://medium.freecodecamp.org/why-End-to-End-testing-is-important-for-your-team-cb7eb0ec1504>> Luettu 14.4.2018.
3. Lönn, Ragnar. 2015. Continuous Testing: What exactly is it? Verkkodokumentti.  
<<https://devops.com/continuous-testing-what-exactly-is-it/>> Luettu 13.4.2018.
4. Tricentis. 2010-2018. What is Continuous Testing. Verkkodokumentti.  
<<https://www.tricentis.com/what-is-continuous-testing/>> Luettu 28.2.2018.
5. Modus. 2016. Protractor Automated Tests Structure. Verkkodokumentti.  
<<https://moduscreate.com/blog/protractor-automated-tests-structure/>> Luettu 14.4.2018.
6. Martin, Fowler. 2012. TestPyramid. Verkkodokumentti.  
<<https://martinfowler.com/bliki/TestPyramid.html>> Luettu 15.4.2018.
7. Fishman, Stephen H. 2016. Testing is Good. Pyramids are Bad. Ice Cream Cones are the Worst. Verkkodokumentti.  
<<https://medium.com/@fistsOfReason/testing-is-good-pyramids-are-bad-ice-cream-cones-are-the-worst-ad94b9b2f05f>> Luettu 15.4.2018
8. Software Testing Fundamentals. 2018. Unit Testing. Verkkodokumentti.  
<<http://softwaretestingfundamentals.com/unit-testing/>> Luettu 17.4.2018.
9. Safari Books Online. 2018. Integration testing. Verkkodokumentti.  
<<https://www.safaribooksonline.com/library/view/spring-batch-essentials/9781783553372/ch09s03.html>> Luettu 1.4.2018.
10. CodeUtopia. 2016. What are Unit Testing, Integration Testing and Functional Testing? Verkkodokumentti.  
<<https://codeutopia.net/blog/2015/04/11/what-are-unit-testing-integration-testing-and-functional-testing/>>
11. Sharma, Lakshay. 2016. What is Manual Testing? Verkkodokumentti.  
<<http://toolsqa.com/software-testing/manual-testing/>> Luettu 15.4.2018.

12. Guru99. 2018. How to Create a Test Plan? Verkkodokumentti.  
<<https://www.guru99.com/what-everybody-ought-to-know-about-test-planning.html>> Luettu 18.4.2018.
13. Guru99. 2018. How to Write Test Cases: Sample Template with Examples? Verkkodokumentti.  
<<https://www.guru99.com/test-case.html>> Luettu 18.4.2018.
14. Tutorialspoint. 2018. Defect Logging and Tracking. Verkkodokumentti.  
<[https://www.tutorialspoint.com/software\\_testing\\_dictionary/defect\\_logging\\_and\\_tracking.htm](https://www.tutorialspoint.com/software_testing_dictionary/defect_logging_and_tracking.htm)> Luettu 8.3.2018.
15. International Software Test Institute. 2018. Automated Software Testing. Verkkodokumentti.  
<[https://www.test-institute.org/Automated\\_Software\\_Testing.php](https://www.test-institute.org/Automated_Software_Testing.php)> Luettu 18.5.2018.
16. Software Testing Help. 2018. Step by Step Guide to Implement Proof of Concept (POC) in Automation Testing. Verkkodokumentti.  
<<http://www.softwaretestinghelp.com/implement-proof-of-concept-poc-in-automation-testing/>> Luettu 18.4.2018.
17. Johnston, Bob. 2018. How Test Automation was Hijacked. Verkkodokumentti.  
<<http://www.critical-logic.com/how-test-automation-was-hijacked/>> 18.4.2018.
18. Roth, Chris. 2015. E2E testing with Angular, Protractor, and Rails. Verkkodokumentti.  
<<https://medium.com/how-we-build-fedora/e2e-testing-with-angular-protractor-and-rails-725fbefb8149>> Luettu 16.3.2018.
19. Bar-Zik, Ran. 2016. Pro tips for E2E protractor tests. Verkkodokumentti.  
<<https://www.linkedin.com/pulse/pro-tips-e2e-protractor-tests-ran-bar-zik/>> Luettu 7.3.2018.
20. Pasala, Ram. 2016. E2E Testing with Protractor, Cucumber using TypeScript. Verkkodokumentti.  
<<https://medium.com/@igniteram/e2e-testing-with-protractor-cucumber-using-typescript-564575814e4a>> Luettu 14.4.2018.
21. Protractor asennus ohje. Verkkodokumentti.  
<<https://www.protractortest.org/#/>> Luettu 14.4.2018.
22. TestCafe. 2012-2018. A node.JS tool to automate end-to-end web testing. Verkkodokumentti.  
<<https://devexpress.github.io/testcafe/>> Luettu 14.4.2018.
23. Cypress. The web has evolved Finally, testing has too. Verkkodokumentti.  
<<https://www.cypress.io/>> Luettu 12.4.2018.

24. NightwatchJS. 2015. Browser automated testing done easy. Verkkodokumentti. <<http://nightwatchJS.org/>> Luettu 12.4.2018.
25. SmartBear Software. 2018. Why Automated Testing? Verkkodokumentti. <<https://smartbear.com/learn/automated-testing/>> Luettu 30.3.2018.
26. Seleniumhq. What is Selenium? Verkkodokumentti. <<https://www.seleniumhq.org/>> Luettu 30.1.2018.
27. Katalon LLC. 2018. FAQs. Verkkodokumentti. <<https://www.katalon.com/faqs>> Luettu 31.1.2018.
28. Grant, Josh. 2018. Watir is. Verkkodokumentti. <<http://watir.com/>> Luettu 30.1.2018.
29. IBM. Rational Functional Tester. Verkkodokumentti. <<https://www.ibm.com/us-en/marketplace/rational-functional-tester/details#product-header-top>> Luettu 28.2.2018.
30. Robotframework. Introduction. Verkkodokumentti. <<http://robotframework.org/>> Luettu 28.3.2018.
31. Eggplant. 2018. AI powered predictive suite transforms testing, speed, delivery, and bug free apps. Verkkodokumentti. <<https://www.testplant.com/2018/02/12/testplant-ai-powered-predictive-suite-transforms-testing-speed-delivery-bug-free-apps/>> Luettu 20.2.2018.
32. Tricentis. Why API Testing? Verkkodokumentti. <<https://www.tricentis.com/api-testing/>> Luettu 20.2.2018.
33. Tricentis. SAP Testing. Verkkodokumentti. <<https://www.tricentis.com/software-testing-tools/sap-testing/>> Luettu 20.2.2018.
34. Ranorex GmbH. 2018. Test Automation for All. Verkkodokumentti. <<https://www.ranorex.com/>> Luettu 15.2.2018.
35. Kiviniemi, Pekka. 2018. Writing Readable Test Automation. Verkkodokumentti. <<https://www.slideshare.net/Qentinel/writing-readable-test-automation-qentinel-automation-clinic-132018>> Luettu 30.11.2017.
36. Aalto, Juha-Markus. 2018. Ecosystem Automation as a Service. Verkkodokumentti. <<https://www.slideshare.net/Qentinel/ecosystem-automation-as-a-service-qentinel-automation-clinic-132018>> Luettu 30.11.2017.