Responsiivinen käyttöliittymien testiautomaatio

Ammattikorkeakoulututkinnon opinnäytetyö

Valkeakosken kampus, Sähkö- ja automaatiotekniikka, Insinööri (AMK)

Kevät, 2021

Tero Jaakkola

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Sähkö- ja automaatiotekniikka | Tiivistelmä
Valkeakosken kampus

Tekijä	Tero Jaakkola	Vuosi 2021
Työn nimi	Responsiivinen käyttöliittymien testiautomaatio	
Ohjaajat	Juha Sarkula	

TIIVISTELMÄ

Tässä opinnäytetyössä toteutettiin käyttöliittymän testiautomaation konseptimalli Profit Software Oy:n asiakasprojektiin. Konseptimallin tarkoituksena on kartoittaa asiakasprojektin testiautomaation tarpeita sekä vähentää sen toteutuksen riskejä. Tämän yhteydessä vertailtiin erilaisia testiautomaatiojärjestelmiä tavoitteena löytää järjestelmä, joka on helposti laajennettavissa ja ominaisuuksiltaan monipuolinen. Testiautomaation tarkoituksena on vähentää testaamisen työpanosta ja nopeuttaa sitä.

Teoriaosuudessa kerättiin onnistuneen testiautomaation rakentamiseen vaadittavaa pohjatietoa. Opinnäytetyössä käytiin läpi mitä testaus on ja minkälaisia tavoitteita testaukselle tulisi asettaa. Testityypit auttavat määrittelemään testien vaatimuksia, jonka avulla pystytään valitsemaan oikeanlainen testausmenetelmä testiin. Testausmallit toimivat korkean tason ohjenuorana testejä suunnitellessa.

Tämän tiedon pohjalta vertailtiin kahta erilaista testiautomaatiojärjestelmää ja tutkittiin niiden ominaispiirteitä ja käyttökohteita. Vertailussa saavutettiin hyvä kuva testiautomaatiojärjestelmien vahvuuksista ja heikkouksista. Konseptimalli toteutettiin onnistuneesti ja raamit lopullisen testiautomaation toteuttamiseksi löydettiin. Tilanhallinta testiautomaatiossa jouduttiin jättämään toteutuksen ulkopuolelle, sillä se vaatii lisää keskustelua ja tutkimustyötä oikean vaihtoehdon löytämiseksi.

Avainsanat Cypress, käyttöliittymän automaatiotestaus, Playwright, testiautomaatio
Sivut 37 sivua ja liitteitä 1 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Electrical and Automation Engineering
Valkeakoski

Abstract

Author	Tero Jaakkola	Year 2021
Subject	Responsive user interface test automation	
Supervisors	Juha Sarkula	

ABSTRACT

In this thesis user interface test automation proof of concept was created for a customer project at Profit Software Oy. The purpose of the proof of concept was to examine the requirements for test automation and to decrease risks in the execution stage. At the same time, a test automation comparison was conducted here to find a system that would be easy to be expanded and that would have a diverse set of features. The purpose of test automation is to decrease the time spent on testing and to increase the test execution speed.

The theory part in this thesis presents prerequisites for conducting successful test automation. The thesis examines testing in detail and what kind of goals should be set when testing. Test types help to determine requirements for tests so that the correct test level can be chosen. Test models operate as a higher-level guideline when designing tests.

Based on this information, two different test automation systems were compared and their features and use cases were studied. A good understanding of the strengths and weaknesses of test automation systems was established. Test automation proof of concept was performed successfully and ground for final test automation was found. State control had to be left outside the test automation proof of concept implementation because it would need a closer discussion and research so that the correct option can be found.

Keywords	Cypress, Playwright, test automation, user interface test automation,
Pages	37 pages and appendices 1 pages

Sisällys

1 Johdanto 1
2 Ohjelmistotestaus 2
2.1 Testaustyypit 3
2.1.1 Lasilaatikkotestaus 3
2.1.2 Mustalaatikkotestaus 4
2.1.3 Toiminnallisuustestaus 4
2.1.4 Ei-toiminallinen testaus 5
2.1.5 Muutokseen pohjautuva testaus 5
2.2 Testikattavuus 5
2.3 Verkko-ohjelmointirajapinnan jäljittely 6
2.4 Testausmallit 7
2.4.1 Testipyramidi 7
2.4.2 Testipokaali 8
3 Testausmenetelmät 9
3.1 Staattinen testaus 10
3.1.1 Staattinen analyysi 10
3.1.2 Katselmointi 11
3.2 Dynaaminen testaus 12
3.2.1 Yksikkötestaus 12
3.2.2 Integraatiotestaus 13
3.2.3 Järjestelmätestaus 14
4 Testiautomaatiojärjestelmät 14
4.1 Selenium 14
4.2 Cypress 16
4.3 Playwright 18
5 Automaatiotestaus projektissa 21
5.1 Lähtötilanne 21
5.2 Suunnittelu 22
5.3 Toteutus 23
5.3.1 Asennus 23
5.3.2 Ensimmäiset testit 25
5.3.3 Virheiden etsiminen 27
5.3.4 Tilanhallinta testeissä 27

5.3.5 Testikoodin generointi 28
6 Yhteenveto 31
Lähteet 35

Kuva 1 Testipyramidi ja testipokaali 9
Kuva 2 Selenium-testikehyksen arkkitehtuuri (Elm, ei pvm) 15
Kuva 3 Cypressin arkkitehtuuri muokattu (Elm, n.d.) 16
Kuva 4 Cypressin asennuspaketin tärkeimmät kirjastot (Cypress.io, ei pvm) 17
Kuva 5 Playwright test runnerin rakenne 21
Kuva 6 Testikehyksien asennusvaiheet 24
Kuva 7 Aloitusnäkymän testit kirjoitettuna Cypressilla 25
Kuva 8 Aloitusnäkymän testit Playwrightilla 26
Kuva 9 Cypressin elementin valitsinfunktio 29
Kuva 10 Lomakenäkymän testit 30
Kuva 11 Playwright Inspector 31

Liitteet

Liite 1 Projektin näkymät

1 Johdanto

Tässä opinnäytetyössä toteutetaan käyttöliittymän testiautomaation konseptimalli Profit Software Oy:n asiakasprojektiin. Toteutuksen yhteydessä tutkitaan uusia käyttöliittymien testiautomaatioratkaisuja. Tarkoituksena on löytää käyttöliittymän testiautomaatiojärjestelmä (engl. test automation framework), joka olisi ominaisuuksiltaan monipuolinen ja responsiivinen eli laajennettavissa muille alustoille. Teoriaosuudessa tutustutaan ensin erilaisiin testaustyyppeihin, testausmalleihin ja testausmenetelmiin, jotta saadaan luotua kuvan testikehykseltä vaadittavista ominaisuuksista.

Verkkosovellusten käyttöliittymäkehitys on nopeatempoista ja uusia teknologioita ja ohjelmakehyksiä ilmestyy tiheästi. Uudet verkkosovelluksien käyttöliittymät kirjoitetaan pääsääntöisesti JavaScriptillä ja React on yleisin verkkosovelluksissa käytetty käyttöliittymäkirjasto (Stack Overflow, n.d.-a; Stack Overflow, 2020). Kehitysnopeuden takia tulevaisuuden ennustaminen on vaikeaa ja alaa täytyy seurata tarkasti, jotta siinä pysyy mukana. Myös uusia käyttöliittymän testikehyksiä on ilmestynyt viime vuosina useita. (Microsoft, n.d.-g; Cypress.io, n.d.-b)

Testaus on äärimmäisen tärkeä osa ohjelmistoprojektia, mutta manuaalinen testaus voi olla tylsää ja itseään toistavaa. Resurssien käytön tulisi olla tehokasta ja manuaalinen testaus vie työtunteja, jotka voitaisiin ohjata muihin tarkoituksiin. Testiautomaatiolla automatisoidaan manuaalista ohjelmistotestausta ja sen tarkoituksena on korvata ihmisen tekemää toiminnallista testausta. Automatisointi tulee kyseeseen, kun testaus on hankalaa, aikaa sitovaa tai se toistuu usein. Jatkuva käyttöliittymän testiautomaatio luo varmuutta kehitettävän ohjelmiston oikeasta toiminnasta ja mahdollistaa ketterän kehityksen menetelmien käytön. (VALA Group Oy, n.d.-a; VALA Group Oy, n.d.-b)

Opinnäytetyössä tutkitaan kahta testiautomaatiojärjestelmää, jotka valikoituvat vertailuun alustavan kartoituksen perusteella. Selenium on alalla yleisesti tunnettu testiautomaatiojärjestelmä, joten se toimii vertailussa referenssiratkaisuna (Stack Overflow, n.d.-b). Cypress ja Playwright ovat melko uusia testiautomaatiojärjestelmiä, joiden arkkitehtuuri poikkeaa tavallisista testikehyksistä (Microsoft, n.d.-g; Cypress.io, n.d.-b).

Vertailussa tutkitaan ominaisuuksia myös asiakasprojektin tarpeiden ulkopuolelta, jotta vertailun tuloksia voidaan käyttää tietolähteenä tulevissa projekteissa.

Testikehyksen valinta riippuu ohjelmistoprojektin tarpeista. Eri testikehyksillä on erilaisia vahvuuksia eri käyttötarkoituksiin ja tästä syystä testiautomaation käyttöönotto vaatii testikehyksien kartoitusta. Onnistuneen testiautomaation käyttöönotolla saavutetaan kustannussäästöjä. Testiautomaation myötä palaute tehdystä työstä nopeutuu, jolloin virheiden korjaus on nopeampaa ja tämän myötä kehitystyö helpottuu. Testiautomaatio antaa luottamusta muutoksien tekemiseen. Jatkuvan automatisoidun regressiotestauksen käyttöönotolla vapautetaan manuaaliseen testaukseen käytettyä aikaa, joka voidaan allokoida muihin tärkeisiin käyttötarkoituksiin. (VALA Group Oy, n.d.-b)

2 Ohjelmistotestaus

Testaus on prosessi, jonka tarkoituksena on varmistaa ohjelmiston oikeanlainen toiminta ja löytää toiminnot, jotka eivät ole suunniteltuja (Myers, Sandler & Badgett, 2011, s. 2). Suoritettavia testejä kutsutaan dynaamisiksi testeiksi ja analysointiin perustuvia testejä kutsutaan staattisiksi testeiksi. Ohjelmiston jokaisen mahdollisen tilan läpi käyminen testaamalla on mahdotonta, sillä kaikkien vaihtoehtojen läpikäyminen olisi liian hidasta ja se vaatisi suuren työpanoksen. Näin ollen se ei olisi taloudellisesti kannattavaa, joten emme pysty varmistamaan ohjelmiston täydellistä toimintaa testaamalla. (Myers, Sandler & Badgett, 2011, s. 5) Testauksen tavoitteena tulisi olla määrällisen testaamisen sijaan testien tuoton maksimointi. (Hass, 2008, s. xxv; Myers, Sandler & Badgett, 2011, s. 10)

Kaikki ihmiset ovat jossain määrin kilpailuhenkisiä. Kilpailuhenkisyys on hyvä ominaisuus, jos se kohdistetaan oikein. Ennen testauksen aloittamista on testeille luotava oikeanlainen tavoite, jotta testien kirjoittamista pystytään käsittelemään oikealta kannalta. Testauksen tarkoituksena ei ole todistaa, että ohjelma on virheetön tai että se suorittaa tarkoitetut toiminnot oikein. Tämä on yleinen virhe, joka johtaa vääränlaisten asioiden testaamiseen, sillä ohjelman todistaminen virheettömäksi johtaa ohjelmistokehittäjän testaamaan kokonaisuuksia, jotka eivät todennäköisesti aiheuta virheitä. Kääntämällä ajatuksen ylösalaisin luomme tavoitteen, joka luo parempia testejä. Testauksen tarkoituksena on todistaa, että ohjelmassa on virheitä. Tämä pieni muutos ajatusmallissa tekee virheiden

löytämisestä tavoitteen, joten ohjelmistokehittäjä tulee todennäköisemmin luomaan tehokkaampia testejä. (Myers, Sandler & Badgett, 2011, ss. 5-6)

2.1 Testaustyypit

Testaustyypin tarkoituksena on auttaa tunnistamaan testin vaatimia edellytyksiä, testitapauksia sekä testidataa. Oikean testaustyypin valinta riippuu useasta eri tekijästä kuten testattavan komponentin tai järjestelmän monimutkaisuudesta, käytettävissä olevista resursseista tai odotetuista virheistä. (International Software Testing Qualifications Board, 2018, s. 56)

Tietyt testaustyypit sopivat vain tiettyihin tilanteisiin ja tietyille testaustasoille, kun taas toiset sopivat käytettäväksi jokaisella testitasolla (International Software Testing Qualifications Board, 2018, s. 56). Testitasot ovat testitoimintojen ryhmiä, joita hallitaan ja käytetään yhdessä. Jokainen testitaso tarvitsee sopivan testiympäristön toimiakseen (International Software Testing Qualifications Board, 2018, s. 30). Tässä opinnäytetyössä käsiteltyjä testitasoja ovat yksikkötestaus, integraatiotestaus sekä järjestelmätestaus.

2.1.1 Lasilaatikkotestaus

Lasilaatikkotestauksessa (engl. white-box testing) testit johdetaan tutkimalla järjestelmän sisäisiä toimintoja. Sisäisiä toimintoja voivat olla esimerkiksi järjestelmän lähdekoodi tai järjestelmän sisäiset työnkulut ja tietovirrat. Testidata lasilaatikkotestauksessa luodaan päättelemällä järjestelmän logiikasta. (Myers, Sandler & Badgett, 2011, ss. 10-11) Lasilaatikkotestejä voidaan kirjoittaa jokaisella testitasolla. (International Software Testing Qualifications Board, 2018, s. 40) Yleisimmät lasilaatikkotestaus tekniikat ovat lausetestaus (engl. statement testing) ja lausekattavuus (engl. statement coverage) sekä päätöstestaus (engl. decision testing) ja päätöskattavuus (engl. decision coverage). Joissakin turvallisuuskriittisissä järjestelmissä saatetaan käyttää kehittyneempiä lasilaatikkotestaus tekniikoita. (International Software Testing Qualifications Board, 2018, s. 60)

Lausetestauksessa testataan ohjelmakoodin suoritettavat käskyt eli lauseet. Lausekattavuus mitataan laskemalla testattujen lauseiden määrä jaettuna kaikkien mahdollisten lauseiden

määrällä. Tämä tulos ilmoitetaan tavallisesti prosenttilukuna. Lausekattavuuden ollessa 100% varmistetaan, että ohjelmakoodin jokainen lause on suoritettu vähintään yhden kerran. Tämä ei kuitenkaan varmista, että päätöksentekologiikka on täysin testattu. Tästä syystä lausetestauksella ei voida saavuttaa päätöstestauksen kattavuutta. (International Software Testing Qualifications Board, 2018, s. 60)

Päätöstestauksessa testataan ohjelmakoodin päätösten tuottamia tuloksia. Päätökset ovat yleensä ehtorakenteita kuten if-komento. Tämän saavuttamiseksi testit seuraavat ehtorakenteiden kaikkia ehtoja. Päätöskattavuus mitataan laskemalla kaikkien mahdollisten ehtolauseiden tuottamien tilojen määrä jaettuna suoritettujen testien kattamien tilojen määrällä. Päätöskattavuuden ollessa 100% varmistetaan, että kaikki ehtorakenteiden tuottamat mahdolliset tilat on testattu. Lausekattavuudesta poiketen tämä kattaa myös ehtorakenteiden epätosien ehtojen tilat. Myös niissä tapauksissa, missä epätosi ehto on implisiittinen. (International Software Testing Qualifications Board, 2018, s. 60)

2.1.2 Mustalaatikkotestaus

Mustalaatikkotestauksessa (engl. Black-box testing) kuvitellaan testattava järjestelmä tai komponentti mustana laatikkona, jonka sisäiseen toimintaan ei ole tarkoitus keskittyä. Sen sijaan huomio tulee kohdistaa tilanteisiin, joissa ohjelmisto ei toimi halutulla tavalla. Tällä lähestymistavalla testidata luodaan käyttämällä ainoastaan ohjelmiston määrittelyä eikä ohjelmiston sisäisten toimintojen tuntemusta käytetä testauksen tukena. (Myers, Sandler & Badgett, 2011, ss. 8-9)

2.1.3 Toiminnallisuustestaus

Toiminnallisuustestaus arvioi funktioita, joita järjestelmän tulisi suorittaa. Funktiot vastaavat kysymykseen ”Mitä järjestelmän tulisi tehdä?”. Toiminnallisuustestaus käsittelee ohjelmiston toimintaa, joten mustalaatikkotestauksen metodeja voidaan käyttää määrittelemään testien edellytykset ja testitapaukset. Toiminnallisia testejä tulisi kirjoittaa jokaisella testitasolla. (International Software Testing Qualifications Board, 2018, s. 39)

2.1.4 Ei-toiminallinen testaus

Ei-toiminnallinen testaus arvioi järjestelmän ominaisuuksia. Ominaisuuksia voivat olla muun muassa käytettävyys, suoritusteho tai turvallisuus. Standardi ISO/IEC 25010 luokittelee ohjelmiston laadulliset ominaisuudet. Ei-toiminnallinen testaus vastaa kysymykseen ”Kuinka hyvin järjestelmä toimii?”. Ei-toiminnallisten vaatimusten testejä tulisi suorittaa jokaisella testitasolla, sillä ei-toiminnallisten virheiden löytäminen myöhäisessä vaiheessa voi olla äärimmäisen vaarallista koko projektin onnistumisen kannalta. Mustalaatikkotestauksen metodeja voidaan käyttää toiminnallisten testien tapaan myös ei-toiminnallisten vaatimuksien testauksen määrittelyssä. (International Software Testing Qualifications Board, 2018, s. 40)

2.1.5 Muutokseen pohjautuva testaus

Regressiotestauksen tarkoituksena on löytää uusien ominaisuuksien myötä tulleita tahattomia sivuvaikutuksia. Uudet ominaisuudet voivat vaikuttaa muiden komponenttien tai järjestelmien toimintaan, jolloin tahattomia sivuvaikutuksia voi syntyä. Tahattomia sivuvaikutuksia voi syntyä myös esimerkiksi käyttöjärjestelmän tai tietokantajärjestelmän päivityksien yhteydessä. Tämän kaltaisia tahattomia sivuvaikutuksia kutsutaan regressioksi. Regressiotestaus voidaan suorittaa kaikilla testaustasoilla. (International Software Testing Qualifications Board, 2018, s. 36)

Varmistustestaus suoritetaan, kun ohjelmistossa ilmennyt vika on korjattu. Tämä voi tapahtua suorittamalla uudelleen testit, jotka eivät menneet läpi kyseisen vian vuoksi. Ohjelmisto tulee tarpeen mukaan testata uusilla testeillä, jotta vian korjaukseen vaadittavat muutokset saadaan testattua. Varmistustestauksella vahvistetaan, että ohjelmistossa alun perin ilmennyt vika on onnistuneesti korjattu. (International Software Testing Qualifications Board, 2018, s.41)

2.2 Testikattavuus

Testikattavuutta voidaan mitata usealla eri tasolla ja yksi taso on lausekattavuus, joka laskee testattujen rivien määrää ohjelmakoodissa (Myers, Sandler & Badgett, 2011, s. 42).

Esimerkiksi, jos funktiossa on kolme mahdollista paluu reittiä ja testisi kattavat niistä yhden, on funktion lausekattavuus 33 prosenttia. Testikattavuus ei ole täydellinen mittaustapa, sillä testejä kirjoittaessa tärkein asia on tutkia funktion käyttötarkoitusta, mitä testikattavuus ei meille kerro. Koska puuttuvien käyttötarkoituksien löytämiseen ei ole automatisoituja työkaluja, voidaan testikattavuusraporttia käyttää tähän tarkoitukseen. Testikattavuusraporttia lukiessa tulee funktion rakenteen sijaan miettiä sen käyttötarkoitusta. Jotta kattamattomat ohjelmakoodin osat saadaan testattua, tulee testikattavuusraporttia lukiessa miettiä, mitä käyttötarkoituksia voidaan testata rakenteellisten ominaisuuksien sijaan. (Dodds, n.d.)

Käyttöliittymiä testatessa ei pidä yrittää saavuttaa 100 %:n testikattavuutta, sillä se johtaa implementointi yksityiskohtien testaamiseen, jonka seurauksena testit hajoavat sovelluksen ohjelmakoodin refaktoroinnin yhteydessä. Implementaatio yksityiskohdan testaamisen tunnistamiseen on yksinkertainen sääntö. Jos loppukäyttäjä ei pysty suorittamaan testiä, testataan implementaatio yksityiskohtaa. (Dodds, 2018)

2.3 Verkko-ohjelmointirajapinnan jäljittely

Verkko-ohjelmointirajapinta ei ole aina käytettävissä, kun testejä suoritetaan. Palomuurit saattavat rajoittaa rajapinnan käyttöä tai palvelin saattaa ottaa yhteyden ulkopuolisten organisaatioiden rajapintoihin, joiden testaaminen on epävarmaa. Verkko-ohjelmointirajapinnan jäljittely nopeuttaa testien kirjoittamista, sillä testit voidaan kirjoittaa ennen oikean toteutuksen valmistumista. (Kankanamge, 2012, ss. 150-151)

XMLHttpRequest-objektia, myöhemmin XHR, käytetään vuorovaikutukseen palvelimen kanssa. XHR mahdollistaa datan hakemisen URL-osoitteesta ilman koko sivun päivitystä, joten dataa voidaan noutaa ilman käyttäjän toimien keskeyttämistä. Nimestään huolimatta XHR pystyy hakemaan XML:n lisäksi myös muun tyyppistä dataa. (Mozilla, n.d.-b)

XHR-kutsut ovat tärkeä osa verkkosovelluksen käyttöliittymän testausta. XHR on käyttäjän ja ohjelmistokehittäjän kannalta hyvä ominaisuus, mutta aiheuttaa haasteita testejä kirjoittaessa. Haasteita aiheuttavat epätietoisuus, milloin lähetetty kutsu on prosessoitu tai

milloin kutsu on palannut palvelimelta. XHR-kutsut voidaan käsitellä testeissä kahdella tavalla. (Mwaura, 2021, luku Understanding XHR requests)

Ensimmäinen tapa on odottaa eksplisiittisesti määritelty aika ennen kuin oletetaan kutsun palanneen palvelimelta. Tämä tekee testeistä epäluotettavia, sillä testit epäonnistuvat, jos kutsu ei ole ehtinyt palaamaan palvelimelta sekä myös hitaita, koska odotusaika on asetettava korkeaksi, jotta kutsu palvelimelta ehtii varmasti palamaan. Jotkin testikehykset osaavat odottaa XHR-kutsun palaamista palvelimelta automaattisesti. Tästä mekanismista on useita etuja kuten esimerkiksi virheviestit ovat kuvaavampia, koska ne tarkentuvat epäonnistuneeseen verkkokutsuun. Verkkokutsujen palauttaman datan testaaminen on myös mahdollista ja lisäksi verkkokutsuille voidaan tehdä tynkiä (engl. stub). Tynkä on testausta varten tehty toteutus ohjelmiston komponentista tai verkkokutsusta, jota kutsutaan oikean komponentin sijaan. (Mwaura, 2021, luku Understanding XHR requests)

2.4 Testausmallit

Testausmallin tarkoituksena on luoda visuaalinen malli eri testausmenetelmille sekä testien määrälle. Testausmallit jakavat testit testausmenetelmien perusteella ja määrittävät kullekin testausmenetelmälle sopivan määrän testejä. (Fowler, 2018)

Testausmallit toimivat hyvänä visuaalisena vertauskuvana ja ne auttavat ajattelemaan testejä eri tasoilla. Koska testausmallit ovat yksinkertaisia toimivat ne hyvänä nyrkkisääntönä testejä suunnitellessa ja ne auttavat rakentamaan helposti hallittavia sekä nopeita testiautomaatioita. (Fowler, 2018)

2.4.1 Testipyramidi

Testipyramidin (engl. Test pyramid) tarkoituksena on ilmaista, miksi ja millaisia testejä tulisi kirjoittaa. Pyramidin leveys osoittaa testien määrää ja korkeus taas osoittaa testien kokoa. Kuvasta 1 voidaan huomata, että pyramidia ylöspäin liikkuessa testien koko suurenee, mutta niiden määrä vähenee. Myös testien luomiseen vaadittava työ sekä varmuus ohjelmiston toimivuudesta kasvavat, kun liikutaan pyramidissa ylöspäin (Dodds, 2019). Testipyramidin mukaan yksikkötestejä tulee olla huomattavasti enemmän kuin järjestelmätestejä. Tähän on

muutamia syitä. Järjestelmätestien luominen on mahdollista vasta, kun ohjelmiston kaikki osa-alueet on saatu rakennettua. Järjestelmätestien suorittaminen on hidasta ja lisäksi niiden suorittaminen julkaisuputkessa (engl. Deployment pipeline) on monimutkaista. (Fowler, 2012; Wacker, 2015)

Integraatiotestit tuottavat lähes saman varmuuden kuin järjestelmätestaus, mutta yksinkertaisemmin. Ne ottavat pienen määrän yksiköitä ja testaavat niitä kokonaisuutena. Integraatiotestit mahdollistavat järjestelmätestejä keskitetymmän testien luomisen, joiden hallitseminen on yksinkertaisempaa. (Fowler, 2012; Wacker, 2015)

Yksikkötestien kirjoittaminen on nopeampaa kuin muiden dynaamisten testien kirjoittaminen, sillä yksikkötestien kirjoittaminen on yksinkertaista ja ne ovat luotettavia. Yksikkötestit eristävät ja kohdentavat virheet muita testausmenetelmiä paremmin. Yksikkötestien suurin puute on, että emme pysty testaamaan, miten erilliset yksiköt toimivat yhdessä. Tästä syystä tarvitsemme korkeamman tason testejä. On kuitenkin suositeltavaa toistaa korkeamman tason testin osoittama virhe yksikkötestillä. Näin saavutamme suuremman varmuuden siitä, ettei virhe palaa. Testipyramidi perustuu ajatukseen, että järjestelmätestit ovat herkkiä hajoamaan, kalliita kirjoittaa sekä hitaita. Alempien tasojen testien tarve pienenee, jos korkeamman tason testit ovat nopeita, luotettavia ja edullisia muokata. (Fowler, 2012; Wacker, 2015)

On hyvä huomioida, että testipyramidi on käsitteenä lähes kaksikymmentä vuotta vanha (Fowler, 2012). Tässä ajassa testaustyökalut ovat kehittyneet huomattavasti. Myös käyttöliittymistä on tullut monimutkaisempia, mikä asettaa uudenlaisia haasteita testaukselle.

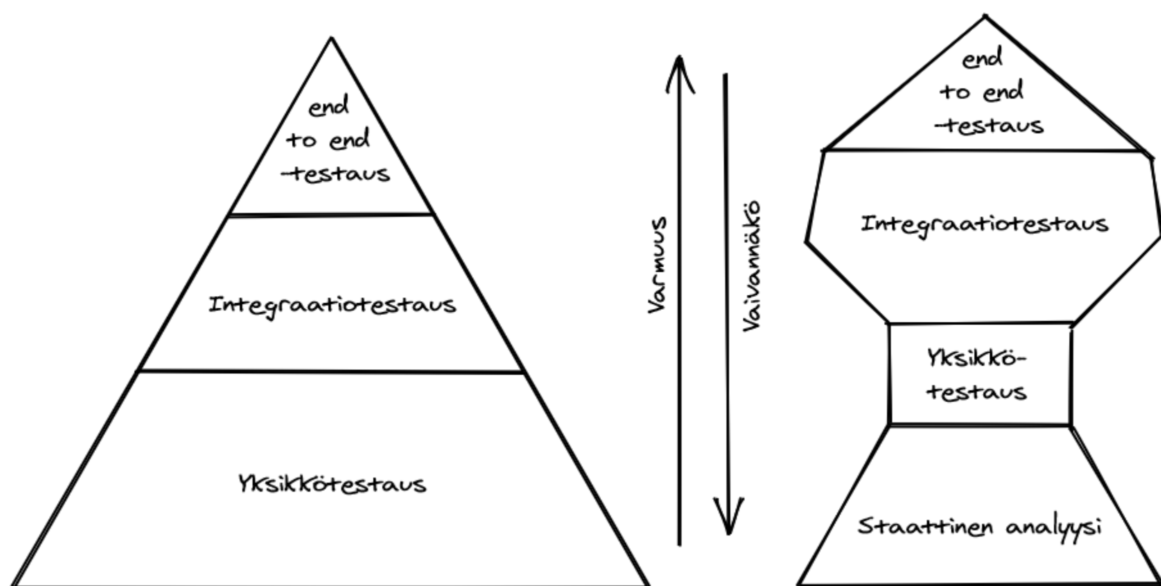
2.4.2 Testipokaali

Vuosien saatossa ohjelmistokehityksessä käytettävien työkalujen tuottama arvo on ylittänyt testipyramidin olettamukset ja tästä syystä Dodds loi testipokaalin (engl. Testing trophy). Testipokaalia luetaan samalla tavalla kuin testipyramidia. Pokaalin leveys osoittaa testien määrää ja korkeus pokaalissa testien kokoa. (Dodds, 2019)

Kuva 1 Testipyramidi ja testipokaali, voimme huomata, että staattinen analyysi on pienentänyt yksikkötestien tarvetta. Tähän on muutama syy. Staattisen analyysin työkalut ovat parantuneet viime vuosina. Työkalut, kuten ESlint ja Typescript, poistavat tarpeen tietynlaisille yksikkötesteille.

Yksittäisten komponenttien testaus ei tuo lisäarvoa, jos komponenttien käyttökelpoisuutta yhdessä ei varmisteta. Jos testaamme useamman komponentin toimintoja, pystymme yleensä testaamaan yksittäiset komponentit riittävällä tasolla. Integraatiotestit pystyvät tarjoamaan hyvän tasapainon niiden tuottaman luottamuksen ja käytetyn työpanoksen suhteen. (Dodds, 2019)

Kuva 1 Testipyramidi ja testipokaali (Dodds, 2019)



3 Testausmenetelmät

Kaikkien testausmenetelmien tarkoituksena on luoda varmuutta sovelluksen oikeanlaisesta toiminnasta sekä löytää virheitä sovelluksesta. Testausmenetelmät voidaan jakaa staattisiin ja dynaamisiin menetelmiin. Staattisessa testauksessa sovellusta ei tarvitse suorittaa, sillä virheet etsitään analysoimalla ohjelmakoodia. Dynaamisessa testauksessa sovellus on suoritettava virheiden löytämiseksi. (International Software Testing Qualifications Board, 2018, s. 47)

Joidenkin virheiden löytäminen on mahdollista vain staattisten menetelmien avulla. Esimerkiksi virheet nimeämiskonventioissa, koodin asettelussa tai ohjelmiston dokumentaatiossa ovat virheitä, jotka pystytään löytämään vain staattisin menetelmin. Osa ajonaikaisista virheistä on mahdollista löytää ainoastaan dynaamisin testausmenetelmin. Virheitä, jotka voidaan löytää sekä staattisin että dynaamisin testausmenetelmin, ovat esimerkiksi virheet muuttujien alkuarvon asettamisessa tai funktion virheellisissä parametreissä. (Homès, 2012, s. 93)

3.1 Staattinen testaus

Staattisessa testauksessa ohjelmistoa ei tarvitse suorittaa virheiden löytämiseksi. Tämä mahdollistaa virheiden löytämisen kehitystyön varhaisessa vaiheessa. Staattisen testauksen tuottoaste on korkeampi, sillä aikaisessa vaiheessa havaittujen ongelmien korjaaminen on edullisempaa ja lisäksi se keskittyy parantamaan ohjelmistotuotteiden sisäistä laatua ja yhdenmukaisuutta. (Homès, 2012, s. 91; International Software Testing Qualifications Board, 2018, s. 47)

Staattinen testaus koostuu katselmoinneista ja staattisesta analysoinnista. Staattinen analyysi on tärkeä osa turvallisuuskriittisiä järjestelmiä, mutta siitä on tullut yleinen testausmenetelmä myös esimerkiksi ketterässä kehityksessä. (International Software Testing Qualifications Board, 2018, s. 46) Katselmoinnit ovat ihmisen suorittamia ohjelmistotuotteen tilan arviointeja (International Software Testing Qualifications Board, n.d.).

3.1.1 Staattinen analyysi

Staattinen analyysi on koodin tai muiden ohjelmistotuotteiden staattista testaamista työkalujen avulla. Rakenteellisten virheiden löytäminen on tärkeä osa staattista analyysia. Rakenteellisia virheitä voivat olla esimerkiksi syntaksivirheet tai virheet ohjelmakoodin asettelussa. Näiden epäkohtien löytämiseksi on useita työkaluja. (International Software Testing Qualifications Board, 2018, s. 46; Homès, 2012, ss. 92-93)

Staattisen analyysin avulla pystymme korvaamaan osan dynaamisista testeistä. Työkalut kuten ESLint ja Typescript mahdollistavat muuttujien tyyppien tarkastukset, joten niitä ei

tarvitse testata dynaamisilla testausmenetelmillä. Staattisen analyysin avulla emme pysty testaamaan liiketoiminnan logiikkaa, vaan sen testaaminen täytyy suorittaa aina dynaamisella testauksella. (Dodds, 2018)

Prettier on yksi staattisen analyysin työkaluista ja se on tarkoitettu ohjelmakoodin muotoilun formatointiin. Prettier toimii täysin automaattisesti, eikä muotoilun asetuksia pysty juurikaan muuttamaan. Tämän ansiosta muotoilun pieniin nyansseihin ei tarvitse tuhlata aikaa. Lisäksi muotoilun formatoinnin virheet versionhallinnassa poistuvat. (Prettier, n.d.)

ESLint on JavaScriptin tarkistustyökalu. JavaScript on dynaamisesti tyypitetty tulkattava kieli, joten se on taipuvainen kehityksen aikaisiin virheisiin. Koska JavaScript-koodia ei käännetä, täytyy koodi suorittaa virheiden löytämiseksi. ESLintin tarkoituksena on mahdollistaa virheiden löytäminen ilman JavaScript-koodin suorittamista. (OpenJS Foundation, n.d.)

TypeScript on vahvasti tyypitetty ohjelmointikieli, joka on rakennettu JavaScriptin päälle. TypeScript tuo muun muassa staattisen analyysin ja käännöksenaikaisen tyyppien tarkastuksen JavaScriptiin. Esimerkiksi väärän tyyppisen parametrin syöttäminen funktioon aiheuttaa virheilmoituksen, koska TypeScript on vahvasti tyypitetty kieli. Koska TypeScript on rakennettu JavaScriptin päälle, on validi JavaScript-koodi myös validia TypeScript-koodia. (Microsoft, n.d.-i)

3.1.2 Katselmointi

Katselmointi on sovelluksen tilan arviointi. Katselmointityypit vaihtelevat vapaamuotoisista muodollisiin. Vapaamuotoisille katselmoinneille ei ole määritelty prosessia ja tulokset voidaan tarvittaessa dokumentoida. Muodollinen katselmointi seuraa tiettyä prosessia ja tuottaa yleisesti vikaraportteja ja katselmointiraportteja. Katselmoinnin painopiste riippuu sille asetetusta tavoitteesta. Tavoitteita voivat olla muun muassa virheiden löytäminen, uusien ideoiden tai ratkaisujen löytäminen tai nimeämiskäytännön yhtenäistäminen. (International Software Testing Qualifications Board, 2018, ss. 48-51)

3.2 Dynaaminen testaus

Staattisilla ja dynaamisilla testeillä voi olla samat tavoitteet, kuten luoda varmuutta sovelluksen toiminnasta ja yrittää havaita virheitä mahdollisimman aikaisin. Dynaamiset testit vaativat sovelluksen suorittamista virheiden löytämiseksi, joten staattiset testit ovat edullisempia suorittaa. (International Software Testing Qualifications Board, 2018, s. 47) Vaikka staattiset testit ovat edullisempia suorittaa eivät ne kykene testaamaan liiketoiminnan logiikkaa, joten osan testeistä on oltava dynaamisia (Dodds, 2018).

Dynaamista testausta voidaan suorittaa eri tasoilla. Näitä tasoja ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus sekä hyväksyntätestaus. (International Software Testing Qualifications Board, 2018, s. 30) Tässä opinnäytetyössä käsittelemämme testaustasot ovat yksikkötestaus, integraatiotestaus sekä järjestelmätestaus. Jokainen testitaso tarvitsee toimiakseen testiympäristön. Testiympäristö voi olla esimerkiksi tuotantoympäristö tai paikallinen kehitysympäristö. (International Software Testing Qualifications Board, 2018, s. 30)

3.2.1 Yksikkötestaus

Yksikkötestin tarkoituksena on testata yhtä toimintoa tai aliohjelmaa eli komponenttia. Yksikkötesti suoritetaan eristettynä muusta järjestelmästä, lukuun ottamatta erikoistapauksia, joissa testin laatiminen ilman ulkopuolista komponenttia on mahdotonta. Komponentti on tavallisesti luokka, funktio tai usean funktion sarja. (Myers, Sandler & Badgett, 2011, s. 85) Yksikkötestaus tunnetaan myös komponenttitestauksena. Automatisoiduilla komponenttien regressiotesteillä on tärkeä rooli ketterässä kehityksessä. Yksikkötestausta käytetään testaamaan toiminnallisia, ei-toiminnallisia sekä rakenteellisia ominaisuuksia. Syitä yksikkötestaukseen on useita. Yksikkötestit voidaan kirjoittaa pienissä osissa ja aikaisessa vaiheessa. Virheiden paikantaminen ja poistaminen yksikkötesteillä on nopeaa, koska löydetyt virheet kohdentuvat yhteen komponenttiin. Yksikkötestien suorittaminen on nopeaa, koska ne voidaan suorittaa rinnakkain. (International Software Testing Qualifications Board, 2018, ss.31-32)

Yksikkötestin kirjoittamiseen vaaditaan komponentin spesifikaatio sekä lähdekoodi. Spesifikaation tulee sisältää vähintään komponentin syöttötiedot, paluuarvo sekä määritelmä komponentin funktiosta. Yksikkötestin vaatimuksista voimme huomata, että yksikkötestaus on suuntautunut lasilaatikkotestaukseen. (Myers, Sandler & Badgett, 2011, s. 86) Yksikkötestin koon kasvaessa siirrytään suorittamaan testin osia peräkkäin, jolloin lasilaatikkotestauksen käyttäminen ei ole käytännöllistä. Tällöin testi täydennetään mustalaatikkotestauksen menetelmien avulla vastamaan komponentin spesifikaatiota. (International Software Testing Qualifications Board, 2018, ss.31-32)

3.2.2 Integraatiotestaus

Integraatiotestit kohdistuvat pääasiassa komponenttien ja järjestelmien väliseen vuorovaikutukseen. Yksikkötestien tapaan automatisoiduilla integraatiotesteillä on tärkeä rooli ketterässä kehityksessä regression ehkäisemiseksi. Integraatiotestien koon kasvaessa virheen kohdentaminen yksittäiseen komponenttiin vaikeutuu ja tämä lisää vian etsimiseen käytettyä aikaa sekä lisää riskiä. Tästä syystä integraatiotestien koko on syytä pitää kohtuullisena. Integraatiotestien pääasialliset tarkoitukset ovat todentaa rajapintojen spesifikaation mukainen toiminta, vahvistaa luottamusta rajapintoihin, löytää virheitä rajapinnoista tai komponenteista sekä estää virheiden eteneminen korkeamman tason testeihin. Integraatiotestit voidaan jakaa kahteen tasoon: komponentti-integraatiotestaukseen ja järjestelmäintegraatiotestaukseen. (International Software Testing Qualifications Board, 2018, ss. 33-34)

Komponentti-integraatiotestaus keskittyy komponenttien välisten rajapintojen ja interaktioiden testaukseen. Ne suoritetaan tavallisesti yksikkötestien jälkeen. Komponentti-integraatiotestit ovat pääosin automatisoituja ja toimivat usein osana jatkuva integraation prosessia. (International Software Testing Qualifications Board, 2018, ss. 32-33)

Järjestelmäintegraatiotestaus kohdistuu pakettien, järjestelmien sekä mikropalveluiden interaktioiden ja rajapintojen testaukseen. Ulkopuolisten toimijoiden rajapintoihin osittain kohdistuvat integraatiotestit ovat osa järjestelmäintegraatiotestausta. Ulkopuoliset rajapinnat tekevät testeistä haastavia, sillä kehityksestä vastaavalla yrityksellä ei ole hallintaa testattavasta rajapinnasta. Lisäksi ulkopuolisten rajapintojen testaukseen vaadittavan

testiympäristön pystyttäminen voi olla vaikeaa. Järjestelmäintegraatiotestaus voidaan suorittaa järjestelmätestauksen jälkeen tai saman aikaisesti järjestelmätestauksen kanssa. (International Software Testing Qualifications Board, 2018, s. 32)

3.2.3 Järjestelmätestaus

Järjestelmätestaus keskittyy koko tuotteen käyttäytymisen ja toiminnallisuuden testaamiseen. Tavoitteet järjestelmätestauksessa ovat pääosin samat kuin integraatiotestauksessa. Järjestelmätestauksessa keskitytään tuotteeseen kokonaisuutena toiminnalliselta ja ei-toiminnalliselta kannalta. Testaus tapahtuu yleensä suorittamalla tuotteen toimintoja ja tarkkailemalla ei-toiminnallisia vaikutuksia tuotteeseen. Alemman tason testien tapaan myös järjestelmätestaus estää regressiota. (International Software Testing Qualifications Board, 2018, ss. 34-35)

4 Testiautomaatiojärjestelmät

Tätä työtä varten tutkittiin kahta eri testiautomaatiojärjestelmää. Alustavina arviointikriteereinä toimivat ohjelmistokehyksen helppokäyttöisyys, ominaisuudet, ohjelmointikieli sekä suosio. Yksi kriteereistä on modernien selaimien tuki, joita ovat Chromium-pohjaiset selaimet, Firefox sekä Safari.

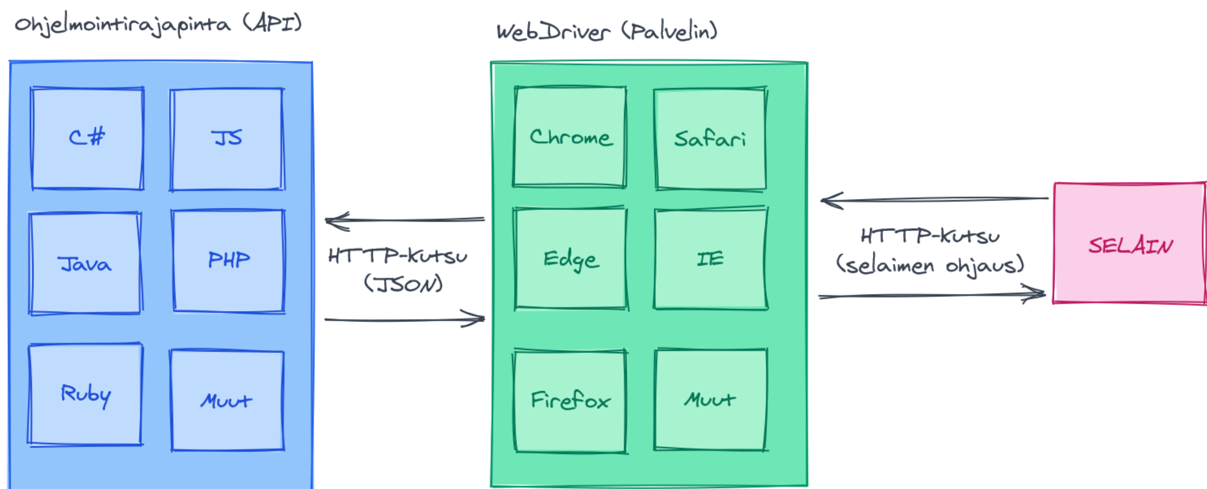
Vertailun testiautomaatiojärjestelmät eroavat ominaisuuksiltaan ja arkkitehtuuriltaan toisistaan, joten tässä vertailussa keskitytään löytämään testiautomaatiojärjestelmälle ainutlaatuisia ja ominaisia piirteitä. Selenium toimii vertailun referenssiratkaisuna, sillä se on alalla yleisesti tunnettu ratkaisu.

4.1 Selenium

Selenium on sarja työkaluja ja kirjastoja, joiden tarkoituksena on automatisoida selaimen käyttöä. Seleniumin ytimessä toimii WebDriver, joka tarjoaa rajapinnan selaimen käytölle. (Software Freedom Conservancy, n.d.-c) WebDriver nimellä viitataan yleisesti ohjelmiston rajapintaan sekä yksittäisen selaimen ajureihin. WebDriver tukee kaikkia merkittäviä selaimia ja ohjaa niitä yhdellä yhtenäisellä rajapinnalla, joka ei ole riippuvainen yksittäisestä

ohjelmointikielestä. WebDriver implementoidaan jokaiselle selaimelle erikseen ja tätä implementaatiota kutsutaan selainajuriksi. Selainajurin tehtävänä on ohjata Seleniumin ja selaimen välistä kommunikaatiota. (Elm, n.d.) Kuva 2 Selenium-testikehyksen arkkitehtuuri voimme tarkkailla Seleniumin rakennetta. Selainajurit ovat pääosin selainvalmistajien valmistamia. (Software Freedom Conservancy, n.d.-b)

Kuva 2 Selenium-testikehyksen arkkitehtuuri (Elm, n.d.)

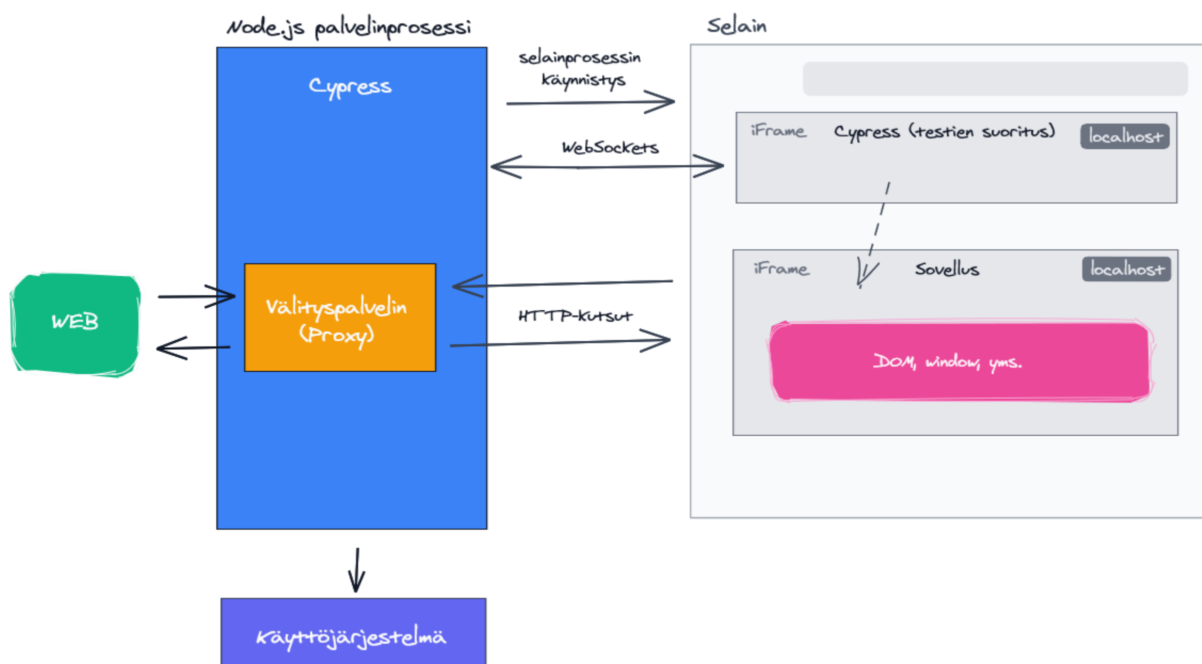


Webdriver on suunniteltu selaimen ohjaamiseen ja sitä käytetään tavallisesti yhdessä testiajo-kirjaston kanssa. Webdriver ohjaa selainta HTTP-kutsuilla selaimen ulkopuolelta, joten sillä ei ole reaaliaikaista yhteyttä selaimen tilaan. Tämä aiheuttaa haasteita, sillä verkkosovellukset ovat luonteeltaan asynkronisia. Esimerkiksi kilpailutilanteet, missä käyttäjän ohjeet eivät vastaa selaimen tilaa, ovat yleisiä. Tästä syystä selaimen ohjausta täytyy kilpailutilanteiden välttämiseksi ohjata eksplisiittisillä odotuksilla. WebDriver odottaa verkkokutsua tai käyttäjätapahtumaa eksplisiittisesti määritellyn ajan tai kunnes odotus-käskyn ehto täyttyy. Ehto voi esimerkiksi odottaa elementin esiintymistä verkkosivulla. Eksplisiittisesti määritelty odotusaika on ongelmallinen, sillä se on määriteltävä riittävän suureksi, jotta se pystyy käsittelemään myös hitaat verkkokutsut. Tämä taas pidentää aikaa, joka WebDriverilla kestää virheen heittämiseksi. Tästä syystä on vaikea määrittää, kuinka pitkään WebDriverin täytyy eksplisiittisesti odottaa verkkokutsua tai käyttäjätapahtumaa. (Software Freedom Conservancy, n.d.-d; Mozilla, n.d.-a; Mwaura, 2021, luku Comparing Cypress and Selenium WebDriver)

4.2 Cypress

Cypress on verkkosovelluksien testiautomaatiokehys. Tavallisesti testiautomaatiokehykset ohjaavat suoritettavia testejä selaimen ulkopuolelta http-kutsujen välityksellä. Cypressissa testit suoritetaan selaimessa, joten sillä on natiivi yhteys selaimen rakenteisiin. Testejä suorittavan osan lisäksi selaimen ulkopuolella toimii palvelinprosessi. Tämän rakenteen ansiosta Cypress pystyy muun muassa seuraamaan ja muokkaamaan verkkokutsuja, kuva 3. Cypressin ohjelmointirajapinta on erityisesti ohjelmistokehittäjille suunnattu ja yksinkertaisen syntaksin ansiosta testien ymmärtäminen on helppoa. (Cypress.io, n.d.-e; Du, 2019; Mwaura, 2021, luku Why choose Cypress?)

Kuva 3 Cypressin arkkitehtuuri muokattu (Elm, n.d.)



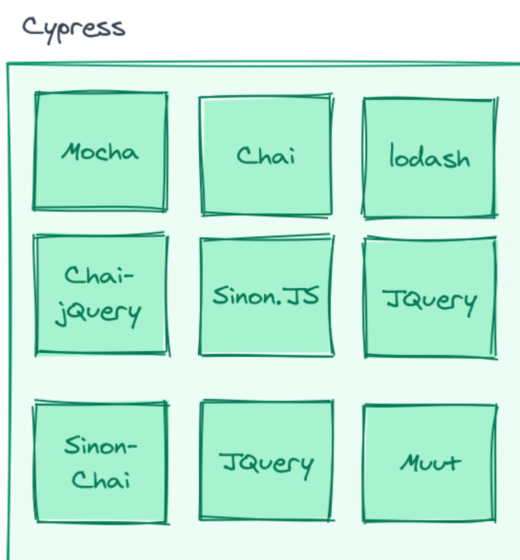
Cypressilla on mahdollista kirjoittaa yksikkö-, integraatio- ja järjestelmätestejä. Testien kirjoittaminen on mahdollista ainoastaan JavaScriptillä. Testattavan sovelluksen ohjelmointikieli tai käyttöliittymäkehys eivät rajoita testausta vaan Cypress voi testata kaikkea, mikä toimii selaimessa. Tuetut selaimet ovat Chromium ja Firefox. Suosituimmat Chromium-pohjaiset selaimia ovat Chrome, Edge ja Opera. (Cypress.io, n.d.-f; Mwaura, 2021, luku Why choose Cypress?, kappaleet 1-5)

Ainutlaatuinen rakenne aiheuttaa haittapuolia. Testejä ei pysty suorittamaan samanaikaisesti usealla selaimen välilehdellä tai selaininstanssilla. Domainin vaihtaminen testissä ei ole mahdollista ja tämä saattaa aiheuttaa ongelmia testattaessa ulkoisia rajapintoja. Ulkoisia rajapintoja voivat olla esimerkiksi maksupalvelut tai kolmannen osapuolen rajapinnat. (Cypress.io, n.d.-d)

Cypress on suunniteltu verkkosovelluksien kehityksenaikaiseksi testiautomaatiojärjestelmäksi. Cypress toimii parhaiten ohjelmistokehittäjän tai laadunvarmistajan työkaluna, jolla kirjoitetaan testit samassa tahdissa kehitettävän ohjelmiston kanssa. Cypress ei ole geneerinen testiautomaatiokehys, joten se ei sovellu esimerkiksi suoritustehotesteihin tai verkkosivuston indeksointiin. (Cypress.io, n.d.-d)

JQuery on osa Cypressin asennuspakettia, kuva 4. JQuery on JavaScript-kirjasto HTML-dokumenttien muokkaukseen ja hallintaan. JQuery tarjoaa yksinkertaisen rajapinnan HTML-dokumenttien hallintaan testeissä. Cypress tarjoaa useita JQueryn metodeja käytettäväksi testikomennoissa. Testikomennot yhdistetään toisiinsa Promise-ketjulla. Cypress hallitsee Promise-ketjua, eikä testaajan ole tarpeen käyttää tätä mekanismia suoraan. (Cypress.io, n.d.-c)

Kuva 4 Cypressin asennuspaketin tärkeimmät kirjastot (Cypress.io, n.d.-a)



Cypress pystyy seuraamaan selaimen toimintoja ja odottamaan kunnes vaadittu toiminto on valmis. Tämä ominaisuus tekee testikutsuista vakaita ja immuuneja useissa testikehyksissä esiintyville vioille. Testikutsut osaavat esimerkiksi odottaa, että DOM on latautunut, verkkokutsu on palannut palvelimelta tai animaatio on suoritettu loppuun. Cypress paketoi kaikki testikutsut logiikkaan, joka yrittää toteuttaa epäonnistuneen testikutsun, kunnes aikaraja tulee vastaan. Aikarajan pystyy muuttamaan joko globaalisti tai kutsukohtaisesti. Aikakatkaisua ei kannata määrittää tarpeettoman pitkäksi, vaikka testien suorittaminen jatkuu välittömästi ehdon täytyttyä, sillä epäonnistuessaan testi odottaa aikakatkaisuun asti ja testien epäonnistuminen kestää tarpeettoman pitkään. Cypress toimii asynkronisesti ja odottaa sovelluksen siirtymistä oikeaan tilaan aikakatkaisuun asti. Tämä tekee testeistä varmatoimisia ja nopeita, koska testikutsuun ei tarvitse määrittää staattista aikarajaa. (Cypress.io, n.d.-c)

Testikutsuja voidaan rakentaa hakemalla HTML-elementtejä tai tekstiä käyttäen JQueryn valitsinmoottoria (engl. selector engine). Elementtien hakeminen on JQuerysta poiketen asynkronista, joten tämä on otettava huomioon testejä kirjoittaessa. Haetun elementin oikea tila voidaan varmistaa väittämä-komennolla (engl. assert). Väittämän avulla voidaan varmistaa elementtien oikea tila ja sisältö. Väittämän avulla voidaan esimerkiksi varmistaa, että elementti on näkyvissä tai että elementillä on tietty tekstisisältö. Cypress odottaa, että väittämän tila vastaa sovelluksen tilaa tai heittää virheen aikakatkaisun jälkeen. (Cypress.io, n.d.-c)

4.3 Playwright

Playwright on Cypressin tapaan keskittynyt verkkosovelluksien testaustarpeisiin. Cypressista poiketen Playwrightin asennuspaketti ei sisällä testiajo-kirjastoa. Playwright on selaimen toimintojen automatisointiin tarkoitettu kirjasto. Ensimmäinen kehitysversio kirjastosta julkaistiin helmikuussa 2020. Playwright on Microsoftin kehittämä kirjasto ja se on kirjoitettu TypeScriptillä. Playwright toimii kahdessa eri ympäristössä. Playwright-skriptit suoritetaan selaimen ulkopuolella ja Page-skriptit suoritetaan selaimessa. Tuettuja selaimia ovat Chromium, Firefox sekä WebKit. Playwrightissa on mahdollista testata kaikki tuetut selaimet käyttäen samaa ohjelmointirajapintaa. Testien suorittaminen päättömänä eli ilman graafista käyttöliittymää on mahdollista kaikilla tuetuilla selaimilla. Kirjastossa on sisäänrakennettuna

emulointi mobiililaitteille, paikannukselle ja käyttöoikeuksille. Playwright varmistaa Cypressin tapaan elementtien valmiuden toiminnoille, mutta toteutustapa poikkeaa Cypressin testikutsu logiikasta. (Microsoft, n.d.-g; Microsoft, n.d.-l)

Playwrightia kehittävä ydinryhmä on aikaisemmin työskennellyt Puppeteer ekosysteemin parissa. Puppeteer on kirjasto, jonka avulla on mahdollista ohjata Chromium-pohjaisia selaimia ja se on suosittu vaihtoehto testattaessa käyttöliittymiä. Ydinryhmä työskentelee nykyisin pääosin Microsoftilla. (Schmitt, n.d.)

Koska Playwrightin rakenne muistuttaa Cypressin rakennetta, on siinä paljon samoja edistyneitä ominaisuuksia. Käyttäjän simuloidut toiminnot osaavat odottaa, että elementti on näkyvillä ja valmis käytettäväksi. Tällaisia toimintoja ovat esimerkiksi hiiren painallus ja tekstikentän täyttäminen. Verkkokutsujen jäljittely on mahdollista sieppaamalla verkkokutsu ja muokkaamalla sitä halutunlaiseksi. Tämän lisäksi Playwrightin rakenne mahdollistaa rinnakkaisajon ja testien ulottumisen usealle eri sivulle, domainille tai iframelle. (Microsoft, n.d.-a; Microsoft, n.d.-e; Microsoft, n.d.-j)

Playwright tarjoaa rajapinnan selaimen ohjaamiseen. Asennuspaketti ei sisällä testiajo-kirjastoa, joten se täytyy asentaa erikseen, jotta verkkosovelluksien testaaminen on mahdollista. Ohjelmointirajapinta on toteutettu usealle ohjelmointikielelle. Tuettuja ohjelmointikieliä ovat JavaScript, Python, C# sekä Java. JavaScriptiä käytettäessä on huomion arvoista, että TypeScript tuki on sisäänrakennettuna testikehykseen eikä erillisiä tyyppikirjastoa tarvitse ladata. Playwright voi etsiä elementtejä CSS-valitsimien, HTML-attribuuttien tai tekstin perusteella. HTML-attribuutteja ovat esimerkiksi id- tai data-attribuutit. (Microsoft, n.d.-d; Microsoft, n.d.-k; Microsoft, n.d.-b)

Selaininstanssin käynnistäminen voi olla kallis toimenpide, joten Playwright on suunniteltu käyttämään yhtä instanssia useassa eri selainkontekstissa. Yksittäinen selainkonteksti on kuin inkognito-tilassa käynnistetty selain istunto. Selainkontekstin käynnistäminen ja käyttäminen on edullista ja yksi selainkonteksti voi sisältää useita sivuja. Sivu on yksittäinen välilehti tai ponnahdusikkuna. Playwrightin rakenne mahdollistaa erilaisten laitteiden emuloinnin sekä testien rinnakkaisajon. (Microsoft, n.d.-b)

Playwright tukee useita suosittuja testiajo-kirjastoja. Tuettuja testiajo-kirjastoja ovat Jest, Jasmine, AVA sekä Mocha. Testiajo-kirjaston asentaminen erillisenä kirjastona ilman valmiita asetuksia lisää testikehyksen käyttöönoton kompleksisuutta. Playwrightin kehittäjiltä on tulossa vakaa versio Playwright test runner –testikehyksestä, joka yksinkertaistaa käyttöönotto prosessia. (Microsoft, n.d.-h)

Playwright test runner on Cypressin kaltainen testikehys, jonka asennuspaketti sisältää kaiken tarvittavan testien kirjoittamiseen ja suorittamiseen. Testikehyksestä ei ole vielä tarjolla vakaata versiota ja toiminnallisuuksia rikkovia ominaisuuksia saattaa tulla ennen ensimmäistä vakaata versiota. Tästä syystä testikehyksen käyttäminen tuotannossa ei ole suositeltavaa. Playwright test runner -testikehys, jonka rakennetta voimme tarkkailla Kuva 5 Playwright test runnerin rakenne käyttää selaimen automatisointiin Playwright-kirjastoa. Cypressista poiketen Playwright test runner ei käytä olemassa olevia testiajo-kirjastoja, vaan käyttää testien ajamiseen Playwright test runneria varten kehitettyä Folio-testikehystä. (Microsoft, n.d.-f)

Folio on Microsoftin kehittämä helposti räätälöitävä testikehys, jonka pääasiallinen tarkoitus on toimia pohjana uusille testikehyksille. Folio eroaa tavallisesta Behavior-driven development testaustavasta luomalla testi fikstuureja. Testi fikstuurin tarkoituksena on luoda jokaiselle testille ympäristö, joka sisältää testille välttämättömät asiat. Testien organisoiminen helpottuu testi fikstuurien myötä. Testejä ei tarvitse ryhmitellä niiden tarvitseman ympäristön perusteella, vaan ne voidaan järjestää testattavan toiminnon perusteella. (Microsoft, n.d.-c)

Kuva 5 Playwright test runnerin rakenne

5 Automaatiotestaus projektissa

Automaatiotestausratkaisujen valinta projektissa riippuu suuresti projektin koosta, aikataulusta sekä projektin rakenteesta. Suurissa ja pitkäkestoisissa projekteissa testiautomaatiolta vaaditaan usein ominaisuuksia, joihin kaikki testiautomaatiojärjestelmät eivät sovellu. Jos esimerkiksi projektissa olisi tarve suorittaa testejä usealla eri sivulla tai välilehdellä samanaikaisesti, ei Cypress soveltuisi projektin testiautomaatiojärjestelmäksi.

Raskaiden geneeristen testiautomaatiojärjestelmien pystyttäminen pieniin projekteihin ei ole järkevää, sillä helposti asennettavan ja käyttöönotettavan testiautomaatiojärjestelmän avulla pystytään luomaan riittävä varmuus sovelluksen toiminnasta pienemmällä työpanoksella. Pienissä projekteissa rajalliset resurssit pakottavat myös priorisoimaan testejä. Testipokaalin perusteella tulisi käyttöliittymäsovelluksessa priorisoida integraatiotestejä, jotka antavat parhaan tuottoasteen tehdylle työlle.

5.1 Lähtötilanne

Asiakasprojekti koostuu käyttäjärajapinnan tarjoavasta käyttöliittymäsovelluksesta sekä verkko-ohjelmointirajapinnan tarjoavasta palvelinsovelluksesta. Palvelinsovellus suorittaa suurimman osan liiketoimintalogiikan vaatimista laskennoista. Asiakasprojektin

käyttöliittymäsovellus on tyypillinen React-käyttöliittymäkirjastoa käyttävä sovellus. Osa siirtymistä näkymien välillä tehdään URL-osoitteita muokkaamalla, mutta sovelluksen tilaa ei pystytä määrittämään URL-osoitteen avulla, mikä on otettava huomioon kirjoitettaessa testejä. Käyttöliittymäsovelluksen tilanhallinta on toteutettu Redux-kirjastolla, joka on suosittu tilanhallintakirjasto. Projektin kannalta tärkeimmiksi testeiksi koettiin regressiotestaus ja kriittisimpien toimintojen varmistaminen. Testeihin käytettävän ajan ollessa rajallinen tavoitetaan regressiotestauksella uusien toimintojen tekemiseen vaadittava varmuus. Projektin siirtyessä tuotantoon on erityisen tärkeää testata sovelluksen toiminnan kannalta kriittisimmät ominaisuudet. Käsittelemme esimerkeissä asiakasprojektin kolmea eri näkymää, Liite 1.

Asiakasprojektin yksityiskohtia ei haluta julkaista, joten asiakasprojektin toteutuksen vaiheita esitetään sitä mukailevin esimerkein. Asiakasprojektin testaussuunnitelmasta valikoidaan erilaisia testitapauksia, jotka toteutetaan mahdollisuuksien mukaan Cypressilla ja Playwrightilla. Asiakasprojektin testiautomaatio tullaan toteuttamaan Cypressilla, joten suuri osa toteutusvaiheen työmäärästä käytetään testien kirjoittamiseen Cypressilla. Esimerkkien avulla saadaan tärkeää tietoa tulevien projektien testiautomaatioratkaisujen kartoittamiseen.

5.2 Suunnittelu

Testisuunnitelma sisältää testejä, joiden tavoitteena on varmistaa sovelluksen oikeanlainen toiminta niin oikein käytettynä kuin virhetilanteissa. Testien kehitysvaiheessa testit suoritetaan headful-tilassa, jotta testien virheiden etsiminen helpottuu. Myöhemmin käytettäessä testejä regressiotestaukseen, tullaan testit suorittamaan headless-tilassa. Headless-tilassa testien suorittaminen on nopeampaa ja mahdollistaa testien suorittamisen julkaisuputkessa.

Aloitusnäkymän testeillä varmistetaan kyseisen näkymän oikeanlainen toiminta. Hakukentän validointi testataan korrektilla ja virheellisellä käyttäjänimellä. Tällä varmistetaan, että hakukentän virheviesti ilmoitetaan ja siirtymää ei tapahdu. Lisäksi jokaisen näkymän testit testaavat siirtymät mahdollisiin seuraaviin näkymiin.

Listanäkymän testaus aloitetaan varmistamalla, että uuden lomakkeen luominen on mahdollista. Lomakelistan oikeanlainen generointi varmistetaan kirjoittamalla tynkä verkkokutsulle, joka noutaa olemassa olevat lomakkeet. Tällä tavoin voidaan varmistaa, että listan tiedot muodostuvat oikein.

Suurin osa testeistä suoritetaan lomakenäkymässä, koska se sisältää suurimman osan käyttöliittymäsovelluksen liiketoimintalogiikasta. Lomakenäkymään on siirryttävä aina listanäkymän kautta, joten sovelluksen tilan vakioimiseksi on kartoitettava mahdollisuuksia muokata sovelluksen tilaa sekä tietokannan sisältöä.

5.3 Toteutus

Cypressin ja Playwrightin testit toteutettiin omaan haaraansa asiakasprojektin versionhallintaan. Koska Cypress tullaan ottamaan käyttöön lopulliseen toteutukseen, käytettiin sen toteutuksen suorittamiseen suurin työpanos. Asennuspakettien asennukseen käytettiin npm-asennustyökalua, joka on JavaScriptille tehty asennustyökalu.

Toteutusvaiheessa kirjoitettiin erilaisia testejä tarkoituksena kartoittaa testiautomaatiojärjestelmien soveltuvuus niihin sekä käytiin läpi testiautomaatiojärjestelmän asennus sekä ensimmäiset testit. Tämän jälkeen keskityttiin tutkimaan testiautomaatiojärjestelmien edistyneempiä ominaisuuksia.

5.3.1 Asennus

Asennuksen tukena käytettiin testikehyksien “Getting started” –sivuja. Molempien testikehyksien asentaminen oli yksinkertaista ja suoraviivaista ja testikehyksien komentorivityökalun käyttö oli hyvin dokumentoitu. Cypress loi valmiiksi hakemistorakenteen ja sisälsi esimerkkitestejä.

Playwrightin testiajo-kirjastona oli ensin tarkoitus käyttää kehitysvaiheessa olevaa Playwright Test Runneria, mutta tämä vaihdettiin ensimmäisiä testejä kirjoittaessa Jestiin, sillä Playwright Test Runnerin dokumentaatiossa ja toteutuksessa oli huomattavia puutteita. Playwrightissa ensimmäisten testien kirjoittamiseen vaadittavat toimenpiteet olivat

dokumentoitu hyvin, mutta dokumentaatio ei ottanut kantaa tiedostorakenteeseen. Jotta testikehyksien käyttö olisi mahdollisimman helppoa, luotiin package.json tiedostoon komennot testien suorittamiseksi. Kuva 6 Testikehyksien asennusvaiheet voimme tarkastella testikehyksien asennusvaiheen toimenpiteitä. Playwright Test Runnerin asennus oli suoraviivainen, mutta voimme huomata, että vaihdettaessa testiajo-kirjasto Playwright Test Runnerista Jestiin asennuksen vaatimat toimenpiteet lisääntyivät, vaikka kyseessä on hyvin yksinkertainen asennus.

Kuva 6 Testikehyksien asennusvaiheet

```
npm install
    playwright
    jest
    jest-playwright-preset
    expect-playwright
    --save-dev

//package.json
{
  "scripts": {
    "jest:run": "jest",
    "jest:debug": "set PWDEBUG=1 && jest",
    "jest:debug:linux": "PWDEBUG=1 && jest",
    "playwright:codegen": "playwright codegen"
  }
}

//.env
SKIP_PREFLIGHT_CHECK=true

npm run jest:run
```

```
npm install cypress --save-dev

//package.json
{
  "scripts": {
    "cypress:open": "cypress open",
    "cypress:run": "cypress run",
  }
}

npm run cypress:run
```

```
npm install @playwright/test --save-dev

//package.json
{
  "scripts": {
    "playwright:run": "npx folio"
  }
}

npm run playwright:run
```

5.3.2 Ensimmäiset testit

Testien kirjoittaminen aloitettiin aloitusnäkymän testeillä. Aloitusnäkymä on näkymistä yksinkertaisin ja sen kriittisimpien toimintojen testaaminen onnistuu muutamalla testillä. Eroavaisuuksia voidaan tarkastella testikehyksien välillä kuvista 7 ja 8. Voidaan huomata, että molempien testikehyksien testit ovat helposti luettavia, mutta Cypress abstraktoi osan testien käyttöönoton vaiheista, jolloin kokonaisuudesta tulee tiiviimpi.

Cypresilla testien kirjoittaminen onnistui vaivattomasti ja testit toimivat luotettavasti. Dokumentaatio oli hyvin järjesteltyä ja yhden sivuston alla. Cypressin ohjelmointirajapinta oli kuvaavaa ja jäljitteli käyttäjän toimintoja ja huomioita, mikä teki testien kirjoittamisesta helppoa. Cypress yrittää väittämän suorittamista aikakatkoon asti ja tämä toiminnallisuus toimi hyvin ja mahdollisti yksinkertaisten testien kirjoittamisen.

Kuva 7 Aloitusnäkymän testit kirjoitettuna Cypressilla

```
const autoRecord = require("cypress-autorecord");

describe("Landing page", function () {
    autoRecord(); // Call the autoRecord function at the beginning of your describe block
    beforeEach(function () {
        cy.visit("http://localhost:3000");
    });

    it("should open", function () {
        cy.contains("Käyttäjän valinta");
    });

    it("should show error on invalid username", function () {
        cy.get(`input[name="userName"]`).type("invalid_username");
        cy.get(`button[type="submit"]`).click();
        cy.contains("Käyttäjätunnus ei ole validi.");
    });

    it("should redirect on valid username", function () {
        cy.get(`input[name="userName"]`).type("demo");
        cy.get(`button[type="submit"]`).click();
        cy.contains("Luo uusi lomake.");
        cy.url().should("include", "/landing?user=demo");
    });
});
```

Playwright Test Runneria käyttäessä ei testien kirjoittaminen käynyt aivan mutkattomasti. Dokumentaatio oli hajautettuna usealle eri sivustolle ja joitain tärkeitä ominaisuuksia ei ollut

dokumentoitu lainkaan. Testien debuggaus ei onnistunut Playwright Test Runnerilla. Tämä ominaisuus oli dokumentoimatta Playwright Test Runnerin GitHub-sivulle, joten testien kirjoittaminen oli hyvin epävarmaa. Tästä syystä päätettiin olla käyttämättä Playwright Test Runneria. Playwright Test Runnerin sijaan otettiin käyttöön Playwright-kirjasto ja Jest-kirjasto erikseen asennettuina. Tämä kokoonpano oli hyvin dokumentoitu ja sen käyttö oli suoraviivaista.

Saatiin huomata, että testiajo-kirjaston vaihtamisen olleen oikea päätös. Jestin avulla testien kirjoittaminen onnistui, mutta niiden kirjoittaminen ei ollut yhtä vaivatonta kuin Cypressillä. Esimerkiksi näkymää vaihdettaessa se piti varmistaa Playwright-kirjaston avulla, koska ”jest-playwright” –kirjaston expect-komento ei osannut odottaa uuden näkymän latautumista.

Kuva 8 Aloitusnäkymän testit Playwrightilla

```
const { chromium } = require("playwright");
const expect = require("expect-playwright");
let browser;
let page;
beforeAll(async () => {
    browser = await chromium.launch({
        headless: false,
        slowMo: 100,
        devtools: true,
    });
});
afterAll(async () => {
    await browser.close();
});
beforeEach(async () => {
    page = await browser.newPage();
    await page.goto("http://localhost:3000");
});
afterEach(async () => {
    await page.close();
});
describe("Aloitusnäkymä", () => {
    it("should fail on invalid username", async () => {
        await page.fill('input[name="username"]', "invalid_user");
        await page.click('button:has-text("Hae käyttäjä")');
        await expect(page).toHaveText("Käyttäjätunnus ei ole validi.");
    });
    it("should redirect on valid username", async () => {
        await page.fill('input[name="username"]', "demo");
        await page.click('button:has-text("Hae käyttäjä")');
        await page.textContent(".landing-search-results");
        await expect(page).toEqualUrl(
            "http://localhost:3000/landing?user=demo"
        );
        await expect(page).toHaveText("Luo uusi Lomake");
    });
});
```

5.3.3 Virheiden etsiminen

Testejä kirjoittaessa testien koodi ei aina ole virheetöntä ja tästä syystä virheiden etsimiseen luotujen työkalujen toiminta sekä käytön helppous ovat erityisen tärkeitä ominaisuuksia. Esimerkiksi testi ei välttämä saa kiinni halutusta elementistä. Tämä voi johtua kirjoitusvirheestä tai ongelmasta testin tai sovelluksen logiikassa. Ilman oikeita työkaluja virheiden paikallistaminen on vaikeaa.

Cypress hoitaa virheiden paikallistamisen erittäin hyvin. Cypress avautuu tavallisesti headful-tilassa ja käynnistää Cypress Test Runnerin. Testien virheiden etsintä, eli debuggaus, Cypress Test Runnerissa oli helppoa ja virheviestit olivat helposti ymmärrettäviä. Erityisen mielenkiintoinen ominaisuus oli aikamatkustus, jonka avulla testin eri vaiheisiin pystyi palaamaan ja näin tarkkailemaan sovelluksen toiminnan muutoksia.

Playwright Inspector ei yllä debuggaus toiminnallisuuksiltaan Cypress Test Runnerin tasolle, mutta virheiden löytäminen oli suoraviivaista. Virheen paikallistamiseksi testin joutui tavallisesti suorittamaan muutamia kertoja. Tämä johtui aikamatkustus-ominaisuuden puutteesta, joten edelliseen tilaan ei ollut mahdollista palata samalla suorituskerralla.

5.3.4 Tilanhallinta testeissä

Testattaessa verkkosovelluksia verkko-ohjelmointirajapinnan käyttö ja sovelluksen sekä tietokannan tilanhallinta on ongelmallista. Testejä kirjoittaessa kaikkia palvelimen ohjelmointirajapintoja ei välttämättä ole vielä toteutettu tai käytetty rajapinta ei sovellu testikäyttöön. Sovelluksen tilan asettaminen halutunlaiseksi asettaa haasteita testiympäristöä pystyttäessä. Testejä suoritetaan useita kertoja, joten tietokannan tilanhallinta on myös huomioitava testiautomaatiota rakennettaessa.

Cypressiin on tarjolla useita kirjastoja verkko-ohjelmointirajapinnan jäljittelyyn. Projektiin valikoitui Cypress Autorecord -kirjasto, joka luo jokaisen verkkokutsun tyngän automaattisesti ensimmäisellä suorituskerralla. Cypres Autorecordin asennus oli helppoa seuraten kirjaston asennusohjeita ja käyttöönottoon vaadittiin ainoastaan yksi funktiokutsu, kuva 7. Cypress Aurotecordin käyttö nopeuttaa testien suorittamista, sillä hitaita verkkokutsuja ei tarvitse kutsua uudelleen jokaisella testiajolla. Verkkokutsujen tynkien

uudelleen luomisen pystyy kytkemään päälle testikohtaisesti tai kaikilta suoritetuilta testeiltä. Tynkien muokkaaminen on mahdollista, sillä ne ovat tallennettuna yksinkertaisina JSON-tiedostoina. Tynkien muokkaaminen on erityisen tarpeellista, kun halutaan luoda uusi ominaisuus, joka muokkaa verkkokutsuja. Tämän kaltaisessa tapauksessa uusi ominaisuus voidaan luoda käyttöliittymään ennen toteutusta palvelimelle. Tynkiä muokatessa on hyvä huomioida, että niiden päivitys ylikirjoittaa manuaalisesti tehdyt muutokset.

Verkko-ohjelmointirajapinnan jäljittely täysin manuaalisesti on hidasta eikä Playwrightille ole tarjolla kirjastoja tynkien generointiin. Tämä ei ole ideaalein tilanne, sillä projektissa kutsutaan kymmeniä päätepisteitä ja ne sisältävät tuhansia parametreja, joten kaikkien päätepisteiden tynkien kirjoittaminen ja ylläpitäminen ei ole mahdollista. Jestille on tarjolla joitain kirjastoja tynkien generointiin, mutta ne eivät toimineet yhdessä Playwrightin kanssa. Tästä syystä lomakenäkymän testit jätettiin kirjoittamatta Playwrightilla.

Sovelluksen tilanhallinnan toteutus jätettiin konseptimallin ulkopuolelle, sillä se vaatii jatkotutkimusta, jotta testien kirjoittaminen jatkossa olisi mahdollisimman helppoa. Yksi tapa sovelluksen tilanhallintaan on muokata sen tilaa Redux-kirjastoa käyttäen. Tämän kaltaisessa tilanteessa ei käyttäisi verkko-ohjelmointirajapintaa, vaan sovelluksen tila asetettaisiin suoraan Redux-storeen. Koska verkko-ohjelmointirajapintaa ei käytetä, ei voida olla varmoja vastaako asetettu tila palvelimen palauttamaa tietoa, mikä vähentää luottamusta testeihin. Toinen tapa on asettaa tietokanta tiettyyn tilaan ennen testien suorittamista. Tämän toteuttamiseksi täytyy päättää, nollataanko tietokannan tila aina ennen testien suorittamista vai luodaanko mekanismi, millä tietokanta saadaan muokattua testikäyttöön sopivaksi.

5.3.5 Testikoodin generointi

Testien kirjoittaminen täysin manuaalisesti on hidasta ja lisää virheiden määrää. Nykyaikaiset testiautomaatioratkaisut tarjoavat työkaluja testikoodin generointiin. Vaikka generoitua koodia ei voitaisi täysin hyödyntää testeissä, nopeuttaa se testien kirjoittamista ja helpottaa elementtien valitsemista.

Cypress Test Runner sisältää interaktiivisen Selector Playground -ominaisuuden, jonka avulla pystymme generoimaan uniikin valitsimen elementille helposti Cypress Test Runnerin käyttöliittymästä. Cypress tarjoaa ohjelmointirajapinnan, jonka avulla on mahdollista muokata logiikkaa, jolla elementin valitsin luodaan. Jotta testeistä saadaan luotettavia ja tarkkoja, on hyvä suunnitella yhtenäinen tapa, jolla elementit valitaan. Tämä valinta pitää tehdä projektikohtaisesti, sillä se riippuu projektin rakenteesta ja käytetyistä kirjastoista.

Tässä projektissa käsitellään suuria lomakkeita, joten yksi elementin helposti yksilöivä tapa on kutsua sitä name-attribuutin perusteella. Tämä on mahdollista, sillä projektissa käytetään muita kirjastoja, jotka varmistavat lomakkeiden kenttien uniikin nimeämisen. Pystymme muokkaamaan Selector Playgroundia valitsemaan elementin name-attribuutin perusteella, jos sellainen löytyy. Jos name-attribuuttia ei ole määritelty, valitaan elementti vakiosääntöjen perusteella. Kuvan 9 funktio sijoitetaan commands.js-tiedostoon commands-hakemistossa eikä muita toimenpiteitä toiminnallisuuden käyttöönottoon vaadita.

Kuva 9 Cypressin elementin valitsinfunktio

```
Cypress.SelectorPlayground.defaults({
    onElement: ($el) => {
        const name = $el.attr("name");

        if (name) {
            return `[name="${name}"]`;
        }
    },
});
```

Elementin valitseminen pelkän name-attribuutin perusteella on hyväksyttävää yksinkertaisissa näkymissä, mutta virheiden mahdollisuus kasvaa näkymän elementtien määrän kasvaessa. Lomakenäkymä sisältää useita lomakkeita, joten valitsimen yksilöllisyyden tarkistamalla varmistetaan, missä lomakkeessa valittu elementti on, Kuva 10 Lomakenäkymän testit. Lomakkeen yksilöimiseen käytetään ”data-cy”-attribuuttia, joka on yleisesti Cypressissa testaukseen käytetty HTML data-attribuutti.

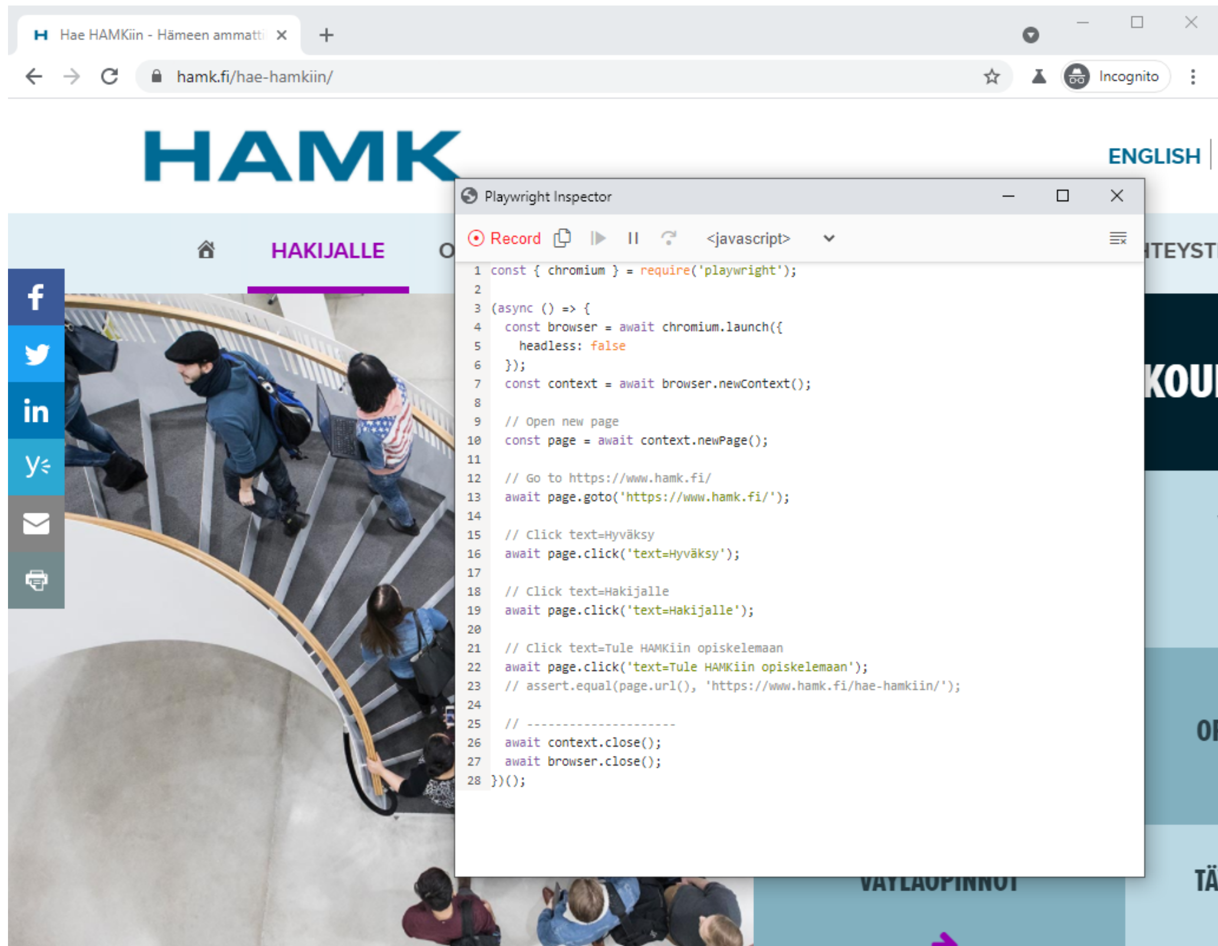
Kuva 10 Lomakenäkymän testit

```
const autoRecord = require("cypress-autorecord");

describe("New form", function () {
    autoRecord();
    beforeEach(function () {
        cy.visit(
            "http://localhost:3000/landing?user=demo"
        );
        cy.get('[href="?user=demo"]')
            .contains("Luo uusi lomake")
            .click();
    });

    it("should open new form", function () {
        cy.url().should("contain", "/form");
        cy.get("form[data-cy='form1']").within(() => {
            cy.get('input[name="user"]').should(
                "have.value",
                "demo"
            );
        });
    });
});
```

Playwright Inspector on työkalu Playwrightilla kirjoitettujen testien tarkkailuun ja debuggaukseen. Playwright inspectorin käynnistäminen testejä suorittaessa onnistuu helpoimmin asettamalla Playwrightin debug-tilan ympäristömuuttuja ennen testikomennon suorittamista, kuten kuvasta 6 voimme huomata. Debug-tila ei toiminut täydellisesti, sillä testien päätyttyä Playwright Inspector sulkeutui ja asetuksia tuon toiminnallisuuden muuttamiseksi ei löytynyt. Ominaisuuksiltaan Playwright Inspector ei yllä Cypressin Test Runnerin tasolle. Myös aikamatkustusominaisuus puuttuu, mikä vaikeuttaa virheiden etsimistä. Playwright Inspectorissa on mahdollista nauhoittaa käyttäjän toimintoja, kuva 11. Käyttäjän toiminnot muutetaan toimivaksi Playwright-skriptiksi. Tämä on hyödyllinen ominaisuus, sillä se mahdollistaa testien alustavan version luomisen ilman kirjoittamista. Playwright Inspectorin luoma skripti on selkeää, tosin joissain tilanteissa se valitsee elementin kestämättömällä tavalla, jolloin testeistä tulee helposti hajoavia. Nauhoitus lisää myös URL-osoitteen muuttuessa ehdotuksen väittämäkäskystä URL-osoitteen muutokselle.

Kuva 11 Playwright Inspector

Playwright Inspector



6 Yhteenveto

Tämän opinnäytetyön tarkoituksena oli toteuttaa testiautomaation konseptimalli asiakasprojektiin ja tämän yhteydessä vertailla ajankohtaisia testiautomaatiojärjestelmiä. Testiautomaatiojärjestelmien vertailuun otettiin kaksi erilaista testikehystä, mutta opinnäytetyön laajuuden rajaamiseksi vertailusta täytyi jättää mielenkiintoisia vaihtoehtoja, kuten Puppeteer pois. Selenium toimi vertailun referenssikehyksenä, jotta uudempien testiautomaatiojärjestelmien ominaisuuksia voitiin vertailla alalla yleisesti käytössä olevaan vaihtoehtoon. Vertailun myötä hahmottui hyvä kuva Cypressin ja Playwrightin vahvuuksista ja heikkouksista. Cypress ja Playwright vaikuttavat pintapuolisesti hyvin samanlaisilta testiautomaatiojärjestelmiltä, mutta syventyessäni niiden dokumentaatioon erot järjestelmien välillä selkenivät.

Cypress on suunniteltu kehityksen aikaiseksi testikehykseksi eikä se sovellu esimerkiksi selaimella suoritettavien työtehtävien automatisointiin. Oikein rajatut ja suunnitellut ratkaisut näkyvät positiivisesti kokonaistuloksessa. JQueryn käyttö käskyketjuissa luo helposti luettavia testejä, vaikka testikehys ei olisi ennestään tuttu. Cypressin ensimmäinen vakaa versio julkaistiin vuonna 2018 (Cypress.io, n.d.-b), joten sen ympärille on ehtinyt kasvamaan suuri yhteisö ja se on suosittu vaihtoehto uusissa React-projekteissa. Cypressin rajoitteet saattavat tulla vastaan erikoisemmissa tilanteissa, missä testien suorittaminen vaatii useamman välilehden tai selaininstanssin testaamista, joita Cypress ei tue. Yksittäinen testi pitää myös suorittaa yhden domainin alla, joten kolmannen osapuolen rajapintojen testaus on ongelmallista.

Playwright on Selenium tapaan selaimen automaatioon tarkoitettu kirjasto eikä sillä ole mahdollista suorittaa laajaa testiautomaatiota ilman erikseen asennettavaa testiajo-kirjastoa. Tämä lisää testiautomaatiojärjestelmän käyttöönottoon vaadittavaa työpanosta. Toisaalta tämän kaltainen rakenne ei rajoita käyttämään testikehyksen valitsemaa testiajo-kirjastoja. Koska Playwright ei sisällä testiajo-kirjastoa, on se mahdollista yhdistää osaksi geneeristä testiautomaatiojärjestelmää, kuten Robot Framework. Robot Frameworkin uusi Browser-kirjasto käyttää Playwrightia selaimen ohjaukseen (Robot Framework, ei pvm). Playwright Inspector vaikuttaa mielenkiintoiselta työkalulta, jonka avulla selkokielisen testikoodin generointi on mahdollista käymällä testitapaus läpi selaimessa.

Uudempia testiautomaatiojärjestelmiä yhdistävä tekijä on, että ne suorittavat testit selaimen sisällä. Tämä on Cypressin esittelemä tapa, joka mahdollistaa aikaisempaa tarkemman selaimen hallinnan ja tarkkailun. Koska selaimessa suoritetuilla testeillä on yhteys kaikkiin selaimen ominaisuuksiin ja tietorakenteisiin, pystytään testaamaan asioita, mitkä eivät ennen olleet mahdollisia. Lisäksi testeistä pystytään selaimessa suoritettuna luomaan tehokkaita ja satunnaiset testien epäonnistumiset vähenevät.

Kumpikaan uusista testikehyksistä ei tue Internet Exploreria, joka saattaa olla näiden vaihtoehtojen valintaa rajoittava tekijä. Modernien selaimien tuki on hyvä molemmilla testikehyksillä, tosin Cypressin WebKit-selainmoottorin tuki on keskeneräinen.

Käytössä testikehyksien erot tulivat selvästi esiin. Playwrightin ekosysteemi ei ole ehtinyt kehittymään samalle tasolle Cypressin kanssa. Cypressin asennus ja käyttöönotto oli suoraviivaista eikä ongelmia esiintynyt. Tämän on tärkeä ominaisuus erityisesti pienissä asiakasprojekteissa, joissa resurssit ovat rajalliset. Mielestäni Playwright ei pysty tämän kaltaisena toteutuksena haastamaan Cypressia kehityksenaikaisena testiautomaatioratkaisuna pienissä tai keskisuurissa projekteissa. On kuitenkin mielenkiintoista nähdä, kuinka helppokäyttöinen ratkaisu Playwright Test Runnerista muodostuu, kun se saavuttaa vakaan version.

Playwright on hyvin yleiskäyttöinen testikehys, joka tukee useita ohjelmointikieliä. Tämä tekee Playwrightista mielenkiintoisen vaihtoehdon, jos käyttöliittymäsovellus rakennetaan jollain muulla kielellä kuin JavaScriptillä. Esimerkiksi Blazor saattaa valikoitua tulevissa projekteissa käyttöliittymäsovelluksen ohjelmistokehykseksi, jolloin Playwrightin mahdollisuutta testikehyksenä tulisi arvioida tarkemmin.

Playwright yhdessä jonkin testiajo-kirjaston kanssa on hyvä valinta testikehykseksi, kun Cypressin tarjoamat ominaisuudet eivät riitä. Esimerkiksi jos projektissa vaaditaan suoritustehotestejä tai verkkosovelluksen käyttöliittymä halutaan testata mobiililaitteilla, on Playwright hyvä valinta. Toisaalta molempien testikehyksien asentaminen on melko yksinkertainen prosessi, joten myös niiden käyttäminen yhdessä on mahdollista.

Asiakasprojektin konseptimallin toteutus eteni suunnitellusti eikä suurempia ongelmia ilmennyt. Sovelluksen ja tietokannan tilanhallinta testeissä osoittautui odotettua suuremmaksi työksi, joten sen toteutusta jouduttiin lykkäämään. Tämä ongelma pitää ratkaista ennen lopullisen toteutuksen aloittamista. Alustavien suunnitelmien perusteella tullaan projektin tietokanta asettamaan testeille sopivaan tilaan. Tietokannan tilan asettaminen oikeanlaiseksi vaatii tarkkaa suunnittelua ja ylläpitoa, mutta parantaa luottamusta testeihin, kun järjestelmä testataan tietokantaan asti.

Olemme opinnäytetyön tilaajan kanssa samaa mieltä, että kaikki tärkeimmät tavoitteet saavutettiin. Tärkeimpänä tavoitteena oli luoda asiakasprojektiin testiautomaation konseptimalli, jonka pohjalta laajempi toteutus voidaan suorittaa. Konseptimallilla pystyttiin luomaan hyvä referenssiratkaisu testien kirjoittamiseksi, jonka pohjalta lopullisen ratkaisun

testeistä pystytään tekemään yhtenäiset. Samalla konseptimallin toteutuksen yhteydessä esiintyneitä ongelmia pystytään ratkaisemaan ajan kanssa, jolloin testiautomaation toteutuksen riskit pienenevät. Testiautomaatiojärjestelmistä muotoutui hyvä kuva, minkä avulla oikean testiautomaatiojärjestelmän valinta tulevissa projekteissa helpottuu.

Lähteet

Cypress.io. (n.d.-a). *Bundled Tools*. https://docs.cypress.io/guides/references/bundled-tools

Cypress.io. (n.d.-b). *Cypress releases*. https://github.com/cypress-io/cypress/releases/tag/v1.0.0

Cypress.io Inc. (n.d.-c). *Introduction to Cypress*. https://docs.cypress.io/guides/core-concepts/introduction-to-cypress.html

Cypress.io Inc. (n.d.-d). *Trade-offs*. https://docs.cypress.io/guides/references/trade-offs.html

Cypress.io. (n.d.-e). *Key Differences*. https://docs.cypress.io/guides/overview/key-differences.html

Cypress.io. (n.d.-f). *Why Cypress?* https://docs.cypress.io/guides/overview/why-cypress.html

Cypress.io, Inc. (n.d.-g). *Launching Browsers*. https://docs.cypress.io/guides/guides/launching-browsers.html

Cypress.io, Inc. (n.d.-h). *Retry-ability*. https://docs.cypress.io/guides/core-concepts/retry-ability.html

Dodds, K. C. (5.3.2018). *Kent C. Dodds – Write tests. Not too many. Mostly integration.* https://youtu.be/Fha2bVoC8SE

Dodds, K. C. (13.7.2019). *Write tests. Not too many. Mostly integration.* https://kentcdodds.com/blog/write-tests

Dodds, K. C. (n.d.). *How to know what to test*. https://kentcdodds.com/blog/how-to-know-what-to-test

Du, N. (8.10.2019). Integration Testing with Cypress. https://youtu.be/YBbf56uQCDI

Elm, D. (n.d.). *Cypress: The future of E2E testing*. https://www.youtube.com/watch?v=pXyBligMMr0

Fowler, M. (1.5.2012). *TestPyramid*. https://martinfowler.com/bliki/TestPyramid.html

Fowler, M. (2018). *The Practical Test Pyramid.* https://martinfowler.com/articles/practical-test-pyramid.html

Hass, A. M. (2008). *Guide to Advanced Software Testing.* Artech House.

Homès, B. (2012). *Fundamentals of software testing.* ISTE/Wiley.

International Software Testing Qualifications Board. (2018). *Foundation Level Syllabus.* https://www.istqb.org/downloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html

International Software Testing Qualifications Board. (n.d.). *ISTQB Glossary.* https://glossary.istqb.org/app/en/search/review

Kankanamge, C. (2012). *Web services testing with soapUI (1st edition.).* Packt Publishing.

Microsoft. (n.d.-a). *Auto-waiting*. https://playwright.dev/docs/actionability

Microsoft. (n.d.-b). *Core conscepts*. https://playwright.dev/docs/core-concepts

Microsoft. (n.d.-c). *Folio*. https://github.com/microsoft/folio

Microsoft. (n.d.-d). *Getting Started*. https://playwright.dev/docs/intro

Microsoft. (n.d.-e). *Network*. https://playwright.dev/docs/network

Microsoft. (n.d.-f). *Playwright test runner*. https://github.com/microsoft/playwright-test

Microsoft. (n.d.-g). *Releases*. https://github.com/microsoft/playwright/releases?after=v0.13.0

Microsoft. (n.d.-h). *Test Runners*. https://playwright.dev/docs/test-runners

Microsoft. (n.d.-i). *TypeScript for the New Programmer*. https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html

Microsoft. (n.d.-j). *Why Playwright?* https://playwright.dev/docs/why-playwright

Microsoft. (n.d.-k). *Why Playwright?* https://playwright.dev/docs/why-playwright

Microsoft. (n.d.-l). https://github.com/microsoft/playwright

Mozilla. (n.d.-a). *Setting up your own test automation environment*. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Your_own_automation_environment

Mozilla. (n.d.-b). *XMLHttpRequest*. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

Mwaura, W. (2021). *End-to-End Web Testing with Cypress.* Packt Publishing.

Myers, G. J.;Sandler, C.;& Badgett, T. (2011). *The Art of Software Testing* (Osa/vuosik. 3). Wiley Publishing.

OpenJS Foundation. (n.d.). *About*. https://eslint.org/docs/about/

Prettier. (n.d.). *Why Prettier?* https://prettier.io/docs/en/why-prettier.html

Robot Framework. (n.d.). *Robot Framework Browser*. https://github.com/MarketSquare/robotframework-browser

Schmitt, M. (n.d.). *What is Playwright?* https://playwright.tech/blog/what-is-playwright

So, C. (2019). Pros and cons of UI testing tools. https://youtu.be/1LK1lV6yCVw

Software Freedom Conservancy. (n.d.-a). *Browsers :: Documentation for Selenium*. https://www.selenium.dev/documentation/en/getting_started_with_webdriver/browsers/

Software Freedom Conservancy. (n.d.-b). *Getting started with WebDriver*. https://www.selenium.dev/documentation/en/getting_started_with_webdriver/

Software Freedom Conservancy. (n.d.-c). *The Selenium Browser Automation Project*. https://www.selenium.dev/documentation/en/

Software Freedom Conservancy. (n.d.-d). *Waits*. https://www.selenium.dev/documentation/en/webdriver/waits/

Software Freedom Conservancy. (n.d.-e). *WebDriver :: Documentation for Selenium*. https://www.selenium.dev/documentation/en/webdriver/

Stack Overflow. (2020). *Stack Overflow Developer Survey 2020*. https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents

Stack Overflow. (n.d.-a). *Stack Overflow Trends*. https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Cangularjs%2Cjquery

Stack Overflow. (n.d.-b). *Stack Overflow Trends*. https://insights.stackoverflow.com/trends?tags=cypress%2Cselenium%2Cpuppeteer

Stephens, R. (2015). *Beginning software engineering.* John Wiley & Sons, Incorporated.

VALA Group Oy. (n.d.-a). *Testiautomaatio*. https://www.valagroup.com/fi/palvelut/testiautomaatio/

VALA Group Oy. (n.d.-b). *Testiautomaatio opas.* https://www.valagroup.com/wp-content/uploads/pdf/Testiautomaatio-opas-2020.pdf

Wacker, M. (2015). *Just Say No to More End-to-End Tests*. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

Liite 1: Projektin näkymät