Ville Järviranta

Greip eService -käyttöliittymän automaatiotestaus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

16.3.2022

Tiivistelmä

Tekijä:	Ville Järviranta
Otsikko:	Greip eService -käyttöliittymän automaatiotestaus
Sivumäärä:	31 sivua
Aika:	16.3.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Head of Technology Aleksanteri Aaltonen Yliopettaja Auvo Häkkinen

---

Insinöörityön aiheena oli suunnitella ja toteuttaa käyttöliittymän automaatiotestaus Greip eService -web-sovellukselle. Työssä esitellään testauksen merkitys, tasot ja lähestymistavat ja perehdytään automaatiotestaukseen hyötyihin ja haittoihin. Työn suunnitteluun ja toteutukseen kuuluvat työvaiheet käydään läpi. Lopuksi arvioidaan työtä ja esitellään mahdollisia jatkokehitysideoita.

Suunnitteluosiossa esitellään, mitä testauksen tavoite oli yrityksen näkökulmasta. Testauksen laajuutta ja testitapausten määrää määritellään. Myös käytettyihin teknologioihin ja testattavaan käyttöliittymään perehdytään. Testaus toteutettiin Robot Framework -automaatiokehystä ja sille kehitettyä Browser-kirjastoa käyttäen. Lisäksi toteutunut automaatiotestaus lisättiin osaksi Azure DevOps -komentoputkea.

Insinöörityön lopputuloksena oli toimiva käyttöliittymän automaatiotestaus, joka on helposti ymmärrettävissä ja sitä on helppo hallita. Toteutus antaa yritykselle tietoa käyttöliittymän mahdollisista regressio-ongelmista ja nopeuttaa kehitystyötä antamalla tarkkoja raportteja testiajoista.

Avainsanat: automaatiotestaus, Robot Framework, Azure Pipelines

Abstract

Author:	Ville Järviranta
Title:	Automation Testing of Greip eService User Interface
Number of Pages:	31 pages
Date:	16 March 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Software Engineering
Supervisors:	Aleksanteri Aaltonen, Head of Technology
	Auvo Häkkinen, Principal Lecturer

The topic of this bachelor's thesis was to design and implement user interface automation testing for the Greip eService web application. The significance, levels, and approaches of testing are presented. The advantages and disadvantages of automation testing are also introduced. The steps involved in the planning and implementation of the study are gone over. Finally, the work is evaluated, and possible further development ideas are presented.

The design part presents what the goal of the testing was from the company's perspective. The scope of the test and the number of test cases are defined. The technologies used and the user interface that was tested are also introduced. The testing was performed using the Robot Framework automation framework and the Browser library developed for it. In addition, the implemented automation testing was added as part of the company's Azure DevOps CI/CD pipeline.

The result of the study was a functional user interface automation testing that is easy to understand and easy to manage. The implementation provides the company with information about possible regression problems in the user interface and speeds up development work by providing accurate reports of test runs.

Keywords: automated testing, Robot Framework, Azure Pipelines

Sisällys

Lyhenteet

CI	*Continuous Integration.* Jatkuva integraatio. Menetelmä lähdekoodimuutosten yhdistämisestä yhteen koodikantaan.
CD	*Continuous Delivery.* Jatkuva julkaisu. Menetelmä sovellusten julkaisun testaamiseen ja hyväksyntään. (Voi myös tarkoittaa *Continuous Deployment.*)
CSS	Cascading Style Sheets. Menetelmä tyylittää web-sivuja.
Git	Avoimen lähdekoodin hajautettu versionhallintajärjestelmä.
HTML	Hypertext Markup Language. Avoimen standardin merkintäkieli, jolla web-sivustot on kirjoitettu.
HTTP	Hypertext Transfer Protocol. Protokolla tiedonsiirtoon web-selainten ja palvelimien välillä.
IPR	Intellectual Property Rights. Immateriaalioikeus, kuten patentti, tavaramerkki tai malli.
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto tiedonvälitykseen.
NPM	Node Package Manager. Node.js-ajoympäristön pakettienhallintatyökalu.
pip	Työkalu, jolla hallinnoidaan Python-paketteja.
UI	*User interface.* Käyttöliittymä. Ohjelmiston osa, jonka kautta käyttäjä käyttää tuotetta.

XPath	XML Path Language. Kieli XML-dokumenttien osien löytämiseen.
YAML	*YAML Ain't Markup Language.* Tiedostoformaatti, jota käytetään usein konfiguraatiotiedostoissa.

1 Johdanto

Insinöörityön tavoitteena oli luoda automatisoitu käyttöliittymätestaus Greip eService -web-sovellukselle käyttäen moderneja teknologioita ja saada se osaksi web-sovelluksen CI/CD (Continuous Integration/Continuous Delivery) -putkea. Tämän lisäksi toteutuksen tavoitteena oli antaa selkeää ja hyödyllistä tietoa web-sovelluksen toimivuudesta. Ohjelmistotestauksen tasot ja yleiset lähestymistavat esitellään yleisellä tasolla, ja automatisoitu käyttöliittymän testausprojekti käydään läpi vaihe vaiheelta. Lisäksi tämän insinöörityön tavoitteena oli selvittää, miten toteutusta voi jatkokehittää ja tehdä arviointi tehdystä toteutuksesta.

Projektissa automaatiotestauksen ytimenä toimii Robot Framework. Robot Framework on automaatiokehys, jota käytetään yleisesti kaikenlaisten automatisoitujen töiden ajoon. Tämä automaatiokehys liitetään osaksi Microsoft Azure CI/CD -putkea, jotta uudet julkaisut web-sovelluksesta regressiotestataan ja että testiajoista generoitu raportti saadaan näkymään Microsoft Azure DevOps -palvelussa. Lisäksi projektissa käydään läpi, mitä pohjustustyötä testaus vaatii, ja käytetyt teknologiat esitellään. Lopuksi tehdyn työn toteutusta arvioidaan sille annettujen tavoitteiden kannalta, ja yhteenvedossa käsitellään, minkälaisia mietteitä ja havaintoja prosessista jäi käteen.

Työ toteutettiin Greip IP Solutions Oy:lle. Greip IP Solutions Oy on vuonna 2018 perustettu yritys, joka kehittää ja ylläpitää Greip-sovellusta. Alun perin Greip-sovellus on kehitetty Berggren Oy:llä, joka on yksi suurimmista patenttiasiamiestoimistoista Suomessa. Vuonna 2018 Greip-sovelluksen kehitys, konsultointi ja liiketoiminta siirrettiin uuden Greip IP Solutions Oy -yrityksen alle. Greip IP Solutions Oy:llä on tällä hetkellä noin 20 työntekijää, ja päätoimisto sijaitsee Helsingin Kampissa. Muita toimistoja sijaitsee myös Oulussa ja Tampereella. Liikevaihto yrityksellä vuonna 2020 oli 830 tuhatta euroa.

2 Ohjelmistotestaus yleisesti

Tässä luvussa käydään läpi ohjelmointitestauksen merkitys ja tavoitteet yleisellä tasolla. Ohjelmointitestauksen yleiset lähestymistavat ja tasot esitellään ja perehdytään automaatiotestauksen merkitykseen osana kehitysprosessia.

2.1 Testauksen merkitys

Jos kehitettävää ohjelmistoa ei testata missään vaiheessa sen elinkaaressa, on vaarana rahallisen arvon menettäminen ja pahimmassa tapauksessa jopa ihmishenkien menettäminen virheellisen toiminnallisuuden takia. [1.] Ohjelmistotestauksen tarkoitus on varmistaa, että kehitettävä ohjelmisto vastaa sille annettuja määrittelyjä ja ettei ohjelmistossa ilmene virheitä eli bugeja. Kun kehittäjät käyttävät asianmukaisia työkaluja ohjelmiston testauksessa, saadaan aikaan mahdollisimman laadukkaita, tehokkaita ja käytännöllisiä ohjelmia. [2.]

Ohjelmistotestauksella on monenlaisia vaikutuksia ohjelmiston kehityksessä. Kaikki nämä vaikutukset tähtäävät siihen, että kehitettävä tuote olisi laadukas mahdollisimman monella tapaa. Kehitysprosessista saadaan kustannustehokkaampi integroimalla testaus olennaiseksi osaksi sitä. Kun ohjelman toiminnallisuus voidaan varmistaa toimivaksi ja määrittelyn mukaiseksi varhaisessa vaiheessa kehitystä, on tämä yrityksen kannalta rahaa ja aikaa säästävää. Yritys ei ole mitään ilman asiakkaitaan ja toimivalla testauksella on helpompi varmistaa asiakastyytyväisyys tuotetta kohtaan. Nykyaikana ohjelmistoissa käsitellään aiempaa enemmän tietoturvallisesti herkkää materiaalia. Tämän takia ohjelmistojen tietoturvallisuutta testaamalla voidaan taata, että se vastaa nykyaikaisia standardeja tietoturvaan liittyen. [2.]

2.2 Testauksen tasot

Ohjelmistotestaus on yleisesti jaettu neljään tasoon, jotka vastaavat laajuudeltaan eri kokonaisuuksia ohjelmistosta. Nämä eri tasot ovat

- yksikkötestaus (engl. unit testing)

- integraatiotestaus (engl. integration testing)
- järjestelmätestaus (engl. system testing)
- hyväksymistestaus (engl. acceptance testing).

Yksikkötestaustasolla keskitytään kaikista pienimpiin osiin ohjelmistoa eli moduuleihin, funktioihin, olioihin tai yksittäisiin riveihin koodia. Yksikkötestauksessa tarkistetaan esimerkiksi, että jokin funktio ottaa vastaan vain tietynlaista syötettä ja että sen tuloste on aina tietynlainen. Kehittäjä, joka on kirjoittanut tällaisen yksinkertaisen toiminnallisuuden, kirjoittaa yleensä myös sen yksikkötestit. Yksikkötestaamisen tavoitteena on parantaa koodin laatua ja havaita nopeasti, jos koodiin tehdyt muutokset rikkovat ohjelmiston perustoiminnallisuutta. On tilanteita, joissa on tarpeellista tehdä testikomponentteja (engl. mock objects), jotka imitoivat jotakin järjestelmän toimintoja. Tällainen tilanne voi tulla vastaan, jos jonkin komponentin toiminnallisuus on riippuvainen toisesta komponentista ja tätä komponenttia ei ole vielä toteutettu. Tällaisesta komponentista voidaan luoda mock-kirjaston avulla testikomponentti, jotta yksikkötestaus voidaan suorittaa perusteellisesti. [3; 4; 5.]

Integraatiotestauksessa käsitellään suurempaa kokonaisuutta, joka muodostuu aiemman tason moduuleista. Vaikka komponentit ovat erikseen testattu yksikkötesteissä, tämä ei takaa, että komponentit toimivat virheettömästi yhdessä. Tästä syystä on tärkeää tehdä testejä, jotka varmistavat, että komponentit toimivat oikein, kun niistä aletaan rakentamaan suurempaa kokonaisuutta. Integraatiotestauksessa voidaan esimerkiksi tarkistaa, että kahden komponentin rajapinnan toiminta keskenään on määritelmän mukaista tai että samaa tietokantaa käyttävien komponenttien yhteistoiminta on käyttökelpoista. On mahdollista, että integraatiotestausvaiheessa on komponentteja, joita ei esimerkiksi ole vielä toteutettu ja näitä komponentteja varten on luotava sijaiskomponentteja, jotka imitoivat jonkin komponentin toiminnallisuutta samalla tavalla kuin yksikkötestauksessa testikomponentti. [3; 4.]

Komponenttien integraatiotestausta voidaan lähestyä kolmella yleisimmällä tavalla, jotka ovat

- alhaalta ylöspäin (engl. bottom up testing)
- ylhäältä alaspäin (engl. top down testing)
- voileipätestaus (engl. sandwich testing).

Kun integraatiotestausta lähestytään alhaalta ylöspäin tämä tarkoittaa sitä, että testaaminen aloitetaan kaikista alimman tason komponenteista eli niistä, jotka ovat lähimpänä esimerkiksi käyttöjärjestelmää tai laitteistoa missä ohjelma ajetaan. Testausta jatketaan lisäämällä komponentteja alempien tasojen päälle, kunnes kaikki järjestelmän toteuttavat komponentit ovat käyty läpi. Tällä tavalla on helppo löytää virheitä matalan tason komponenteista. [3.]

Ylhäältä alaspäin on periaatteeltaan täysin sama kuin alhaalta ylöspäin, mutta integraatiota aletaan testaamaan komponenttien hierarkian ylätasosta alaspäin. Tällä tavalla järjestelmästä saa helpommin paremman yleiskuvan ja puuttuvat toiminnallisuudet löytyvät tehokkaammin. [3.]

Voileipätestauksessa komponenttien integraatiotestausta lähestytään molemmista suunnista samaan aikaan. Tarkoituksen on saada molempien aiemmin mainittujen tapojen hyödyt käyttöön yhtäaikaisesti. [3.]

Kun kaikki komponentit on todettu toimivan keskenään integraatiotestauksessa, siirtyy ohjelmisto järjestelmätestaukseen. Järjestelmätestauksessa validoidaan, että ohjelman määritellyt toiminnallisuudet ja yhteensopivuus ohjelman ja sen ympäristön välillä ovat riittävällä tasolla. Tämä taso on yleensä jaettu kahteen alatasoon, jotka ovat toiminnallinen ja ei-toiminnallinen testaus. Toiminnallisella testauksella tarkoitetaan jonkin ohjelmiston toiminnan testausta. Tällainen toiminnallisuus voisi esimerkiksi olla web-sovelluksen sisäänkirjautuminen. Ei-toiminnallisella testaamisella tarkoittaan ohjelmiston teknillisten vaatimusten validointia. Kuormitustestaus on hyvä esimerkki ei-toiminnallisesta testauksesta. Siinä tarkoituksena on kuormittaa ohjelmistoa, jotta voidaan saada varmuutta, että se toimii vakaasti ennen kuin se julkaistaan todelliseen käyttöön. [3.]

Hyväksymistestaus on viimeinen ohjelmistotestauksen tasoista ja on hyvin samanlainen kuin aikaisempi taso, mutta testaamisen suorittaa sille määritelty käyttäjäryhmä. Tällainen testaus on tarpeellista silloin, kun ohjelmisto on

muuten valmis, mutta sen käyttöä halutaan testata valvotusti ennen kuin se julkaistaan suuremman käyttäjäkunnan käyttöön. Tämä taso on jaettu kahteen alatasoon, jotka ovat alpha- ja beetatestaus. [3; 4.]

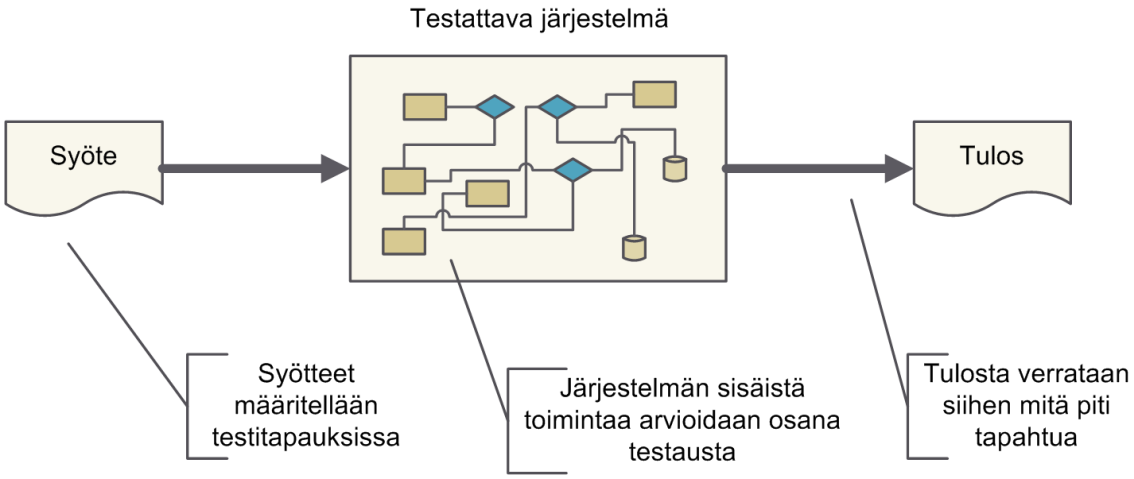
Alphatestaus tapahtuu yleensä sisäisesti yrityksessä siihen valikoidulle ryhmälle. Tälle ryhmälle luodaan skenaarioita, jotka pyrkivät vastaamaan tosi elämän tilanteita ohjelmiston kanssa. Yleensä kehittäjät ovat vahvasti tässä prosessissa läsnä, jotta esimerkiksi ympäristöä, missä ohjelmisto ajetaan, voidaan nopeasti vaihtaa toiseen testauksen aikana. [3; 4.]

Beetatestauksessa ohjelmisto annetaan valittujen loppukäyttäjien käyttöön. Tässä vaiheessa ohjelmisto on yleensä jo valmis ja lähestulkoon vapaa vioista. Käyttäjiä ohjeistetaan antamaan palautetta ohjelman toimivuudesta ja siitä toimiko ohjelma odotetulla tavalla. Erona alphatestaukseen on, ettei tosielämän tilanteita tarvita enää simuloida, vaan annetaan ohjelma käyttöön pienelle ryhmälle todelliseen käyttöön. Kaikki tarvittavat tasot ovat käyty läpi ohjelmistotestauksessa, kun tämä vaihe on läpäisty onnistuneesti ja ohjelma on valmis julkaistavaksi suuremmalle käyttäjäkunnalle. [3; 4.]

2.3 Testauksen lähestymistavat

Kun ohjelmistoa aletaan testaamaan, ei ole tärkeää vain määritellä, mitä osaa tai kokonaisuutta testataan, vaan myös se, mistä näkökulmasta testaaminen tapahtuu. Ohjelmistotestaus on perinteisesti jaettu kahteen lähestymistapaan, joita kutsutaan laatikkomalliksi. Tämä malli sisältää kaksi toisistaan poikkeavaa lähestymistapaa, joita kutsutaan white-box- ja black-box-testaamiseksi. [3.]

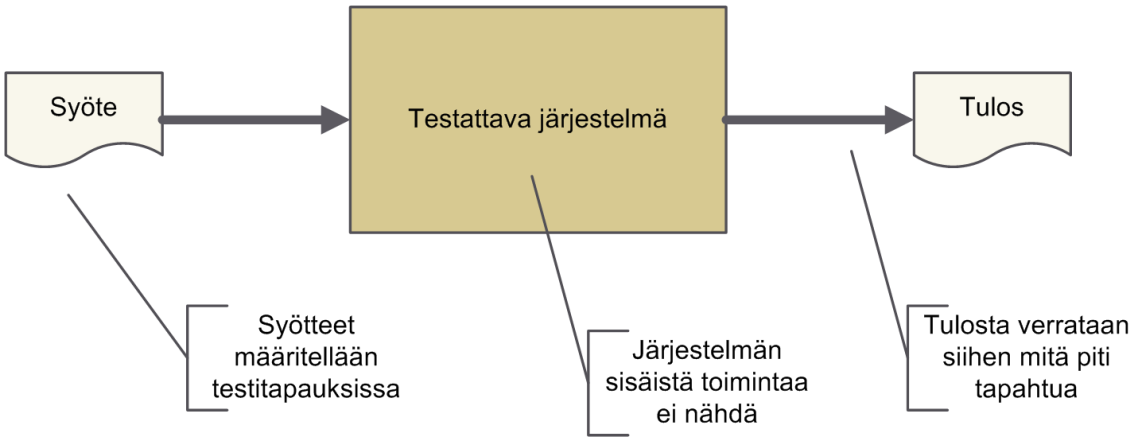
White-box-testaaminen, jota kutsutaan myös rakenteelliseksi testaamiseksi, on sellaista testaamista, missä testaajalla on pääsy ohjelmiston koodiin, joten testaaja on yleensä myös koodin kirjoittaja. Kuvassa 1 näkyy, miltä tallainen white-box-järjestelmä näyttää yleisellä tasolla. [3.]



Kuva 1: White-box-testaus. [3, s. 42]

Ohjelmoijalle on annettu tarkka määrittely siitä, miten koodin pitäisi toimia. Ohjelmoija luo kirjoitetun koodin ympärille testejä, jotka vahvistavat, että koodi käyttäytyy sille annettujen määritelmien mukaisesti. Tällainen testaus tapahtuu yleisesti yksikkötasolla, mutta sitä käytetään myös muissa tasoissa, kuten integraatio- ja järjestelmätasolla, toisin sanoen kaikkialla missä ymmärrys ohjelmiston rakenteesta on tärkeää sen optimaalisen testauksen kannalta. [3.]

Black-box-testaaminen, jota kutsutaan myös toiminnalliseksi testaamiseksi, on sellaista testaamista, jossa testaajan ei ole välttämätöntä tietää, miten ohjelmisto on toteutettu. Kuvassa 2 näkyy, miltä tällainen black-box-järjestelmä näyttää yleisellä tasolla. [3]



Kuva 2: Black-box-testaus. [3, s. 41]

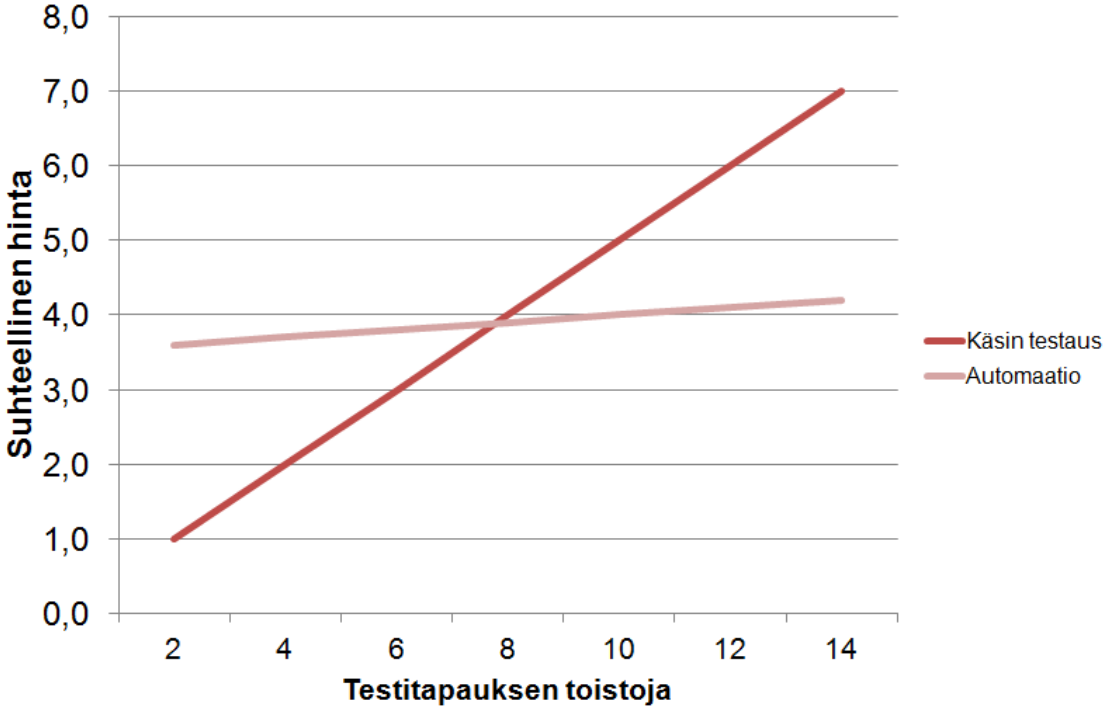
Tällaisessa testaamisessa kyse on enemmän siitä, että ohjelmisto toimii toiminnallisesti, kuten se on määritelty. Esimerkiksi, kun ohjelmasta napsauttaa käyttöliittymästä tiettyä painiketta, tapahtuu tietty asia käyttöliittymässä. Koska testaajan ei tarvitse tietää, miten koodi on kirjoitettu, voi tällaista testausta suorittaa miltei kuka vaan. Tällaista testausta yleensä suorittaa yritysten laadunvarmistuksesta vastaavat yksiköt. [3.]

Näistä kahdesta lähestymistavasta on muodostunut myös kolmas gray-box-testaaminen. Tämä on näiden kahden tavan hybridimuoto, jossa testataan ohjelmisto toiminnallisuutta, mutta puhtaasta black-box-testauksesta poiketen myös koodin rakenne on tiedossa tai osittain tiedossa. [3.]

2.4 Automaatiotestaus

Automaatiotestaus on testaamista, jossa sovelluksen testaaminen tapahtuu jollakin automaatiotyökalulla. Tavoite tällaisella testauksella on vapauttaa kehittäjiä muihin tehtäviin ja antaa automaatiotyökalujen hoitaa päivittäistä tai useasti suoritettavaa testausta. Yleensä automaation kohteina ovat yksikkötestaus, hyväksymistestaus ja käyttöliittymätestaus. Automaatiotestaus ei poista manuaalisen testauksen tarvetta kokonaan, koska automaatiotestauksella lähtökohtaisesti pyritään varmistamaan, ettei jo olemassa olevat ominaisuudet ja toiminnot sovelluksessa hajoa, kun koodiin tehdään muutoksia. Manuaalisella testauksella on todennäköisempää löytää uusia bugeja ja virheitä koodista kuin automatisoidulla testauksessa. [3; 6.]

Muita hyötyjä, joita automaatiotestauksella pyritään saavuttamaan, on tehdä testauksesta kustannustehokkaampaa. Kun sovelluksen automaatiotestaus on saatu siihen pisteeseen, ettei sen kehittämiseen tarvita enää niin paljon työtunteja kehittäjiltä, on testaaminen tällöin paljon halvempaa kuin pelkästään manuaalinen testaaminen. Kuvassa 3 olevassa käyrässä näkyy, että automaatiotestauksessa suhteellinen hinta nousee huomattavasti hitaammin verrattuna käsin tehtyyn testaukseen, kun testitapausten toistojen määrä kasvaa. [3.]



Kuva 3: Automatisoitujen testitapausten kustannuskäyrä. [3, s. 50]

Vaikka automaatiotestauksen tarkoitus on helpottaa testausta saamalla siitä nopeampaa ja halvempaa, on silti mahdollista, ettei automaatiotestauksella saavuteta haluttuja hyötyjä. Näitä ongelmia voivat olla esimerkiksi väärien asioiden testauksen automatisointi, vääränlaisen automaatiotyökalujen käyttö, puutteellinen automaatiotestauksen ylläpito organisaatiotasolla. Automaatiotestauksen käyttöönoton ja ylläpidon haasteellisuudesta huolimatta on selvää, että automaatiotestauksella on merkittäviä vaikutuksia aika- ja kustannussäästöihin, kun automaatiotestaus saadaan toimimaan kunnolla. [3.]

On syytä huomioida, että automaatiotestaus on laadunvalvonnallinen työkalu, eikä testauksen työkalu. Automaatiotestauksen tarkoitus ei ole tutkia ohjelmiston uusien osien toimivuutta, vaan huolehtia, etteivät jo toimivat osat rikkoudu. [3.]

3 Greip eService -käyttöliittymän testauksen suunnittelu

Tässä luvussa käydään läpi käyttöliittymän automaatiotestausprojektin tavoite, käytetyt teknologiat ja suunnittelun työvaiheet. Myös testattavaan käyttöliittymään perehdytään.

3.1 Työn tavoite

Työn tavoitteena oli luoda ensimmäinen versio Greip eService SaaS web-sovelluksen käyttöliittymän automaatiotestauksesta ja tuoda se osaksi kehitysprosessia. Testauksesta haluttiin tehdä mahdollisimman helposti ylläpidettävä ja tehdä siitä myös helposti jatkokehitettävä. Testiajojen tulokset haluttiin saada helposti saataville kehittäjille, jotta mahdollisiin regressio-ongelmiin pystyttäisin reagoimaan mahdollisimman nopeasti. Tämän mahdollistamiseksi testiajon integroiminen osaksi web-sovelluksen CI/CD-putkea varmistaisi näiden tavoitteiden saavuttamista.

3.2 Testattava käyttöliittymä

Testauksen kohteena oleva käyttöliittymä oli Greip eService -immateriaalioikeuksien hallintatyökalun web-sovelluksen graafinen käyttöliittymä. Tärkeimmät käyttöliittymän toteutuksessa käytetyt teknologiat ovat

- TypeScript-ohjelmointikieli
- React-kirjasto
- Material UI -kehys.

TypeScript on Microsoftin kehittämä ja ylläpitämä ohjelmointikieli, joka on yli-joukko JavaScript-ohjelmointikielestä. Sen tärkein muutos JavaScript-kieleen on, että TypeScript on tyypitetty kieli. Tämä tarkoittaa esimerkiksi sitä, ettei funktioille voi antaa mitä tahansa muuttujaa parametriksi ilman, että sille on annettu tyyppi, mikä on täysin mahdollista JavaScript-kielessä. Tämä tekee koodista paljon vähemmän alttiin bugeille ja pakottaa kehittäjien olemaan tarkkoja

siitä, että esimerkiksi rajapinnoista tulevan datan tyypitys on määritelty oikein. [7.]

React on Metan (ennen Facebook) kehittämä ja ylläpitämä JavaScript-kirjasto, jota käytetään web-sovellusten käyttöliittymien luontiin käyttäen komponentteja. Sen toiminnallisuus perustuu näiden komponenttien tilojen hallinnointiin ja näiden tilojen muutosten näyttämiseen itse käyttöliittymässä. React on tällä hetkellä yksi yleisimmistä kirjastoista käyttöliittymien luomiseen. [8; 9.]

Material UI on React-kirjaston ympärille kehitetty kehys, jonka avulla pystyy rakentamaan käyttöliittymiä nopeasti käyttäen valmiiksi tehtyjä React-komponentteja. Tämä kehys sisältää valmiita komponentteja, jotka auttavat muun muassa käyttäjän syötteiden kanssa, datan esittämisessä, palautteen antamisessa käyttäjälle, käyttöliittymän navigoinnissa ja käyttöliittymän asettelussa. [10.]

Greip eService -web-sovelluksen keskeisimmät ominaisuudet ovat erilaiset immateriaalioikeuksien hallinnointiin liittyvien tietojen tarkastelu ja immateriaalioikeuksiin liittyvien ihmisten, kuten patentin omistajan ja patenttiasiamiehen välinen kommunikointi. Testaus liittyi näiden ominaisuuksien toiminnan validointiin ja mahdollisten regressio-ongelmien havaitsemiseen.

3.3 Käytetyt teknologiat

Tässä luvussa käydään läpi tärkeimmät automaatiotestausprojektissa käytetyt teknologiat. Teknologioista annetaan yleiskuva niiden toiminnallisuuksista.

3.3.1 Robot Framework

Robot Framework on avoimen lähdekoodin automaatiokehys, jota käytetään yleisesti testiautomaatiossa ja ohjelmistorobotiikassa (engl. robotic process automation). Sitä kehitetään ja tuetaan aktiivisesti, ja monet ohjelmistokehitysalan yritykset käyttävät sitä kehitysprosesseissaan. Koska Robot Framework on avoin, se on helppo integroida miltei minkä tahansa työkalun kanssa, kun luodaan tehokkaita ja joustavia automaatioratkaisuja. [11.]

Robot Frameworkin syntaksi on yksinkertainen, koska monimutkainen toiminnallisuus on piilotettu avainsanojen taakse. Avainsanat käyttäytyvät kuten metodit ohjelmoinnissa yleisesti. Avainsanoille annetaan yleensä yksi tai useampi argumentti. Jotkut avainsanat palauttavat syötteen, jonka voi tallentaa muuttujaan ja käyttää myöhemmin toisen avainsanan yhteydessä. Esimerkkikoodi 1:ssä on esimerkki hyvin yksinkertaisesta testitapauksesta, jossa kirjaudutaan palvelimeen ja validoidaan kirjautuminen. [11.]

```
Login User with Password
    Connect to Server
    Login User            ironman    1234567890
    Verify Valid Login    Tony Stark
    [Teardown]    Close Server Connection
```

Esimerkkikoodi 1. Testitapaus sisäänkirjautumisen validoinnille.

Robot Framework on helppo laajentaa Python- tai Java-ohjelmistokielellä toteutetuilla kirjastoilla. Kirjastoja kehitetään erillisinä projekteina Robot Frameworkin ympärillä olevien yhteisöjen toimesta. Koska Robot Framework on avoin ohjelmisto, on se myös täysin ilmainen kenen tahansa käytettäväksi ja suurin osa sille kehitetyistä kirjastoista ovat myös täysin avoimia. [11.]

3.3.2 Browser-kirjasto

Browser-kirjasto on Robot Frameworkille kehitetty kirjasto, joka antaa mahdollisuuden automatisoida toimintoja web-selaimissa. Tämä kirjasto antaa avainsanoja Robot Frameworkiin, joilla pystytään kontrolloimaan web-selainta. Tällaisia avainsanoja ovat esimerkiksi selaimen avaaminen, uuden sivun avaaminen ja sivulla olevan painikkeen napsauttaminen. Browser-kirjasto on luotu käyttämään Playwright-testiautomaatiokehystä, joka pyörii Node.js-ajoympäristössä. [12; 13.]

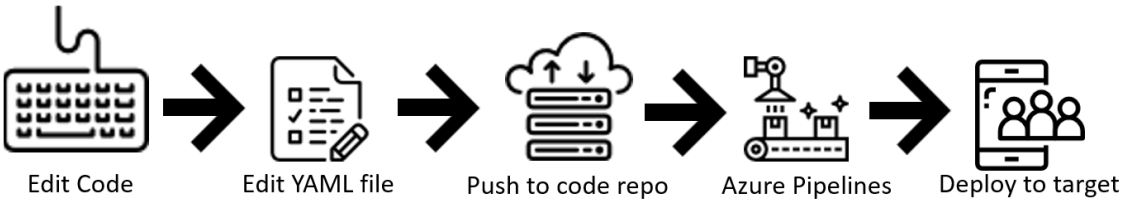
Playwright on Microsoftin kehittämä testiautomaatiokehys, jonka avulla pystyy automatisoimaan toimintoja web-selaimilla. Sillä on mahdollista käyttää kolmea eri selainmoottoria, jotka ovat Chromium, Firefox ja WebKit. Nämä kolme selainmoottoria kattavat yli 85 % kaikista internettiä käyttävistä selaimista. [14; 15.]

Node.js on JavaScript-ajoympäristö, jonka avulla pystytään ajamaan JavaScript-koodia selaimen ulkopuolella. Node.js on suosittu palvelimien luomiseen web-sovelluksia varten. Node.js:n mukana tuleva pakettienhallintaohjelma nimeltä NPM (Node Package Manager) tekee Playwright-kehyksen käyttöönotosta huomattavasti helpompaa. [16.]

3.3.3 Azure Pipelines

Microsoftin Azure DevOps [17] -palvelussa on mahdollista luoda komentoputkia (engl. pipelines), joiden tarkoituksena on automatisoida erilaisia operaatioita, jotka ovat tärkeitä sovellusten kehityskaaressa. Tällaisia toimintoja ovat esimerkiksi sovelluksen testaaminen, koonti (engl. build) ja julkaisu (engl. release). Näitä putkia kutsutaan koonti- ja julkaisuputkiksi (CI/CD). [18.]

CI/CD [19] ovat molemmat menetelmiä, joita käytetään DevOps [20] -toimintamallissa. Kun esimerkiksi sovelluksen koodiin tehdään muutoksia, jatkuva integrointi (CI) vastaa siitä, että nämä muutokset yhdistetään automaattisesti koodikantaan. Jatkuvajulkaisu (CD) vastaa siitä, että sovellukseen tehdyt muutokset päivitetään esimerkiksi testiympäristöön, missä kehittäjät voivat nähdä, miten sovellus käyttäytyy. Kuvassa 4 näkyy, miten Azure Devops -palvelun CI/CD-putket toimivat yleisellä tasolla. [19; 21.]



Kuva 4: Azure-julkaisuputken toiminta yleisellä tasolla. [18]

Näiden putkien toiminta perustuu putkien käynnistykseen jonkin laukaisijan (engl. trigger) toimesta. Tällainen laukaisija on yleensä koodin lisäys tiettyyn haaraan koodikannassa. Tämän jälkeen putkeen määritellyt toimet käydään läpi, yleensä haaraa vasten, mihin koodimuutoksia on tehty. Näitä toimia voivat olla esimerkiksi ohjelmiston testaus, koonti ja julkaisu. [21.]

Jotta tällaisen putken pystyy lisäämään Azuren DevOps-palveluun on koodikantaan lisättävä YAML (YAML Ain't Markup Language) -tiedosto. Tämä tiedosto sisältää muun muassa tiedot siitä, mitä toimintoja putken täytyy suorittaa ja mikä aiheuttaa putken käynnistymisen. [21.]

3.4 Testauksen laajuuden määrittely

Kaikenlaisessa testauksessa on tärkeää määritellä tarvittavat testitapaukset, jossa kerrotaan

- mitä testataan
- miten testaus tapahtuu
- mikä on oletettu tulos.

Jos testattava ominaisuus on esimerkiksi sovelluksen sisäänkirjautumisen validointi, testitapauksessa pitää ilmetä kaikki vaiheet, miten sovellukseen kirjaudutaan sisään ja miten sisäänkirjautuminen validoidaan onnistuneeksi. [3.]

Yrityksessä oli tehty jo töitä käyttöliittymän testaamisessa tarvittavien testitapausten laatimisessa ja näistä testitapauksista oli myös valittu sovelluksen toiminnallisuuden varmistamisen kannalta tärkeimmät. Nämä tärkeimmät testitapaukset liittyivät sovelluksen ydinominaisuuksiin, jotka voidaan jakaa kahteen kategoriaan

- tietojen tarkasteluun
- tietojen lisäämiseen.

Koska projektin tavoitteena ei ole tehdä kokonaisvaltaista automatisoitua käyttöliittymäntestausta, vaan alustaa automaatiotestaus osaksi kehitysprosessia, oli näiden tärkeimpien testitapausten valinta projektin laajuudeksi luontevaa. Automaatiotestauksesta saataisiin siten nopeasti tietoa sovelluksen olemassa olevien päätoimintojen regressiosta. Testitapauksia, joita projektiin otettiin mukaan, oli kaiken kaikkiaan 20 kappaletta. Huomioitavaa näissä testitapauksissa on myös se, ettei niissä missään keskitytty virhetilanteisiin eli testitapaukset käsittelevät vain sovelluksen oikeaoppisen käytön toimivuuden varmistavista.

3.5 Testidatan luonti

Testitapausten määrittelyn jälkeen vuorossa oli testidatan luonti. Greip-web-sovelluksessa käsitellään immateriaalioikeuksiin liittyvää dataa, ja tämä on hieman ongelmallista testauksen näkökulmasta. Testaukseen ei voi käyttää minkään yrityksen oikeaa IPR-portfoliota, koska siinä piilee tietoturvariskejä. Greip IP Solutions Oy:llä oli olemassa aitoa dataa sisältävä testidata, missä kaikki arkaluonteinen tieto oli muutettu satunnaiseksi tekstiksi, ettei se ollut enää ihmisen ymmärrettävissä eikä takaisinmuunnettavissa. Tämä data soveltuu hyvin manuaaliseen testaukseen, mutta koska data oli muodoltaan niin sekavaa, olisi sen käyttö ollut haastavaa automaatiotestauksessa. Ainoa vaihtoehto oli, että testidata luotaisiin itse automaatiotestausta varten.

Koska datan ei tarvinnut olla validia IPR-näkökulmasta, testidataa pystyi luomaan suhteellisen helposti. Dataa luotiin tietojen tarkasteluun tarvittava määrä, johon kuului paljon tyhjiä patentteja, joissa oli rajallisesti tietoa mukana. Nämä tyhjät patentin olivat tarpeellisia, koska ne antoivat dataan tarvittavaa kokoa, joka näkyisi käyttöliittymässä numeraalisena tietona. Myös tarvittavia työtiloja alustettiin testidataan, jotta tarpeellisiin osiin käyttöliittymää olisi pääsy tietoja muokkaavia testitapauksia varten. Työtilat ovat sivuja käyttöliittymässä, joissa viestittely ihmisten välillä tapahtuu sovelluksessa.

Käytännössä tämä prosessi toteutettiin käynnistämällä web-sovelluksen palvelintaso asettamalla se käyttämään tyhjää tietokantaa. Kun palvelintaso käynnistyy, se samalla alustaa tyhjän tietokannan ja antaa sille oikean rakenteen. Tämän jälkeen tietokantaan pystyi lisäämään dataa. Datan lisääminen tehtiin käyttämällä web-sovellukseen tehtyä rajapintaa, minkä tarkoitus on siirtää dataa tietokantaan ja sieltä ulos. Tämä rajapinta käyttää HTTP-protokollaa tiedonsiirrossa ja siirrettävä data annettiin JSON-muodossa rajapintaan, josta se siirtyi edelleen tietokantaan.

3.6 Web-sovelluksen alustus

Jotta Robot Frameworkilla pystyy automatisoimaan web-sovelluksessa tehtyjä toimenpiteitä, on sen pystyttävä löytämään web-sovelluksen sisältä löytyviä HTML-elementtejä. Nämä elementit voivat olla esimerkiksi painikkeita, tekstikenttiä tai linkkejä.

Projektissa käytetty Robot Framework Browser -kirjasto antaa mahdollisuuden löytää HTML-elementtejä web-sovelluksesta monella tapaa. Yksi näistä tavoista on käyttää CSS-luokkia elementtien etsintään. Tämä tapa vaatii, että kaikilla elementeillä, joita tarvitaan testauksessa, on oltava uniikki tunniste. Jos tätä ei ole otettu huomioon jo varhaisessa vaiheessa web-sovelluksen kehitystä, voi tästä koitua paljon lisätyötä. Toinen mahdollinen tapa löytää HTML-elementtejä on käyttää niiden XPath-tietoa. XPath antaa tiedon, missä kohdassa tietty elementti sijaitsee web-sovelluksen HTML-hierarkiassa. Tässäkin tavassa piilee ongelma. HTML-elementin XPath-tieto muuttuu, jos web-sovelluksen sivun HTML-komponentin sijaintia muuttaa. Tällaisessa tilanteessa pitäisi aina varmistaa, että mahdollinen muutos HTML-elementtien XPath-tiedossa viedään myös Robot Frameworkin testitapauksiin, mutta tämäkin on työlästä.

Edellä mainittujen ongelmien minimoimiseksi projektissa päädyttiin käyttämään tapaa löytää web-sovelluksen HTML-elementtejä antamalla niille uniikit data-* -attribuutit. Data-* -attribuutti on HTML5-versiossa oleva ominaisuus, joka antaa mahdollisuuden lisätä kaikille HTML-elementeille tietoa, joka näkyy HTML-tiedostotasolla. [22.] Tällä tavalla voi antaa HTML-elementeille data-test-id-attribuutin. Browser-kirjasto pystyy löytämään HTML-elementtejä myös näiden data-test-id-attribuuttien avulla.

Tämä tapa on parhain tapa löytää elementtejä web-sovelluksesta, koska vain tarvittaville HTML-elementeille täytyi antaa tämä attribuutti. Kun tämä uniikki arvo on kerran annettu, se ei enää muutu. Ei haittaa, vaikka elementtien sijainti HTML-hierarkiassa muuttuisi, koska tämä attribuutti on liitetty suoraan tarvittavaan HTML-elementtiin. React-komponenteille on myös mahdollista antaa tämä data-test-id-attribuutti, koska React-komponentit kääntyvät aina lopulta HTML-

komponenteiksi. Esimerkkikoodi 2:ssa on TitleInput-nimiselle React-komponentille annetut attribuutit, joissa näkyy myös data-test-id -attribuutti.

```
<TitleInput
    data-test-id="EditableTitle"
    name="title"
    inputRef={register}
    startAdornment={prefix}
    endAdornment={<EndAdornment />}
    readOnly={readOnly}
    disabled={isLoading}
    error={Boolean(errors.title)}
    multiline
    title={title}
/>
```

Esimerkkikoodi 2. TitleInput-komponentille annetut ominaisuudet ja attribuutit.

Joissain tapauksissa tätä data-test-id-attribuuttia ei pystynyt kovakoodaamaan komponenttiin, koska näitä komponentteja saattoi olla monta samaan aikaan käyttöliittymässä. Näissä tapauksissa attribuutti jouduttiin antamaan dynaamisesti komponenteille käyttäen jotain komponentissa esitettävää dataa, jotta komponentille annettava attribuutti olisi uniikki.

Käyttöliittymässä käytetty Material UI -kehys tuotti joissain tapauksissa ongelmia, kun data-test-id-attribuuttia yritettiin lisätä suoraan esimerkiksi tekstikenttään, joka oli upotettu automaattiseen tekstintäydentämiskomponenttiin. Attribuutin lisääminen tällaiseen tekstikenttään aiheutti, ettei tekstintäydentäminen enää toiminut ollenkaan. Tämä ongelma kierrettiin laittamalla attribuutti johonkin ylemmän tason React-komponenttiin. Tarkennusta vaadittiin testeihin, joissa käytettiin tällaisia elementtejä, joihin ei voinut suoraa liittää data-test-id-attribuuttia. Se miten tämä toteutettiin, käydään läpi luvussa 3.7.2.

Koska testattavan käyttöliittymän toteutus ja koodi ovat tiedossa, voi todeta, että testaus toteutettiin white-box-lähestymistavalla.

4 Greip eService -käytöliitymän testauksen toteutus

Tässä luvussa käydään läpi käyttöliittymän automaatiotestauksen toteutus vaiheittain läpi.

4.1 Alkutoimet

Sen jälkeen, kun testitapaukset projektiin oli valittu ja testitapauksissa tarvittaviin HTML-elementteihin oli lisätty data-test-id-attribuutti, aloitettiin itse testiautomaation rakentaminen. Tämä prosessi aloitettiin asentamalla kaikki testauksessa tarvittavat ohjelmistot ja riippuvuudet. Aluksi ladattiin Python, koska Robot Framework toimii Pythonilla ja Robot Framework ladataan käyttäen Pythonin mukana tulevalla pakettienlataustyökalulla nimeltä pip. Myös Browser-kirjasto ladattiin käyttäen pip-työkalua. Viimeinen ohjelmisto, joka täytyi asentaa, oli Node.js-ajoympäristö, koska Broswer-kirjaston käyttämä Playwright-kehys tarvitsee sen toimiakseen. Jotta testejä pystyy ajamaan, oli tarpeellista alustaa Browser-kirjasto komennolla, joka latasi web-selainmoottorit tietokoneelle.

Kun kaikki testauksessa tarvittavat ohjelmistot oli asennettu, luotiin testauksen päätiedosto. Tämä päätiedosto sisältää kaikki Robot Frameworkin tarvitsemat asetukset (engl. settings), testauksessa käytetyt muuttujat (engl. variables) ja itse testitapaukset (engl. test cases). Tämän tiedoston asetuksiin lisättiin tarvittavia kohtia kuten Browser-kirjaston lisääminen testausympäristöön ja mitä alku- ja lopetustoimia tarvittaisiin ennen jokaista testitapausta. Tässä vaiheessa testausympäristöön lisättiin myös Robot Frameworkin sisäinen DateTime-kirjasto, joka sisältää avainsanoja päivämäärien käsittelyssä. Myös tarvittavat muuttujat alustettiin omaan Variables-kohtaan päätiedostossa. Esimerkkikoodi 3:ssa näkyy, miltä päätiedoston asetukset ja muuttujat näyttivät tässä vaiheessa projektia.

```
*** Settings ***

Library       Browser
Library       DateTime
Test Setup    Open new browser   url=${APP_URL}  engine=${BROWSER}
              headless=False     slowMo=1
Test Teardown Close browser

*** Variables ***

${BROWSER}     chromium
${APP_URL}     http://localhost:3000
${AC1_USER}    user_1337
${AC1_USER_PW}password123!
```

Esimerkkikoodi 3. Päätiedoston asetusosio.

Open new browser -avainsana, mikä näkyy esimerkkikoodi 3:ssa, on itse tehty avainsana, joita pystyy lisäämään päätiedostossa omaan kohtaansa. Tässä vaiheessa projektia alkoi jo olla selvää, ettei kaikkea testauksessa tarpeellista tietoa pystyisi sisällyttämään päätiedostoon sellaisella tavalla, että se olisi helposti hallittavissa ja samalla helppolukuista. Tämän ongelman ratkaisemiseksi itsetehdyille avainsanoille tehtiin oma tiedosto. Robot Framework antaa mahdollisuuden käyttää resurssitiedostoja, joissa pystyy säilyttämään testauksessa tarpeellisia itse tehtyjä avainsanoja. Tämä tiedosto lisättiin testausympäristöön Resource-komennolla päätiedoston asetusosiossa.

4.2 Valitsijat

Browser-kirjaston käytössä tarvitaan valitsijoita (engl. selectors), joita käytetään, kun halutaan manipuloida jotain HTML-elementtiä käyttöliittymässä automaatiotestaustarkoituksessa. Web-sovellukseen tehtiin tarvittavia muutoksia, jotta haluttua tapaa identifioida elementtejä käyttöliittymästä olisi mahdollista, kuten luvussa 3.6 kerrotaan. Näitä valitsijoita oli projektissa yli 75 kappaletta.

Jotta kaikkia näitä valitsijoita olisi helpompi hallinnoida päätettiin siirtää ne omaan tiedostoon. Tämä tiedosto lisättiin ryhmänä muuttujia testausympäristöön päätiedoston asetusosiossa Variables-komennolla. Näitä muuttujia käytettiin testitapauksissa valitsijoina sen sijaan, että testitapauksissa olisi suoraan kirjoitettu data-test-id-attribuutin arvo. Tällä tavalla testauksen rakenteesta tuli paljon selkeämpi, koska kaikki valitsijat olivat yhdessä paikassa. Jos valitsijoita

tarvitsee esimerkiksi jostain syystä tulevaisuudessa muuttaa voi muutoksen tehdä tähän valitsijatiedostoon. Tällöin muutosta ei tarvitse tehdä aina jokaiseen testitapaukseen, missä valitsijaa käytetään. Esimerkkikoodi 4:ssa näkyy osa valitsijatiedoston muuttujista.

```
…
LCStatusOpen = "data-test-id=LifeCyclePhase1"
LCStatusDone = "data-test-id=LifeCyclePhase2"
LCStatusUOI = "data-test-id=LifeCyclePhase5"
LCStatusClosed = "data-test-id=LifeCyclePhase4"
NewTaskTitle = "data-test-id=NewTaskTitle"
NewTaskDueDate = "data-test-id=NewTaskDueDate"
NewTaskRelatedField = "div[data-test-id=\"NewTaskRelatedField\"] > div
> input"
…
```

Esimerkkikoodi 4. Ote valitsijatiedoston muuttujista.

Joihinkin valitsijoihin jouduttiin lisäämään tarkentavia tietoja, jotta testitapauksessa tarvittava HTML-elementti löytyisi. Tästä syystä esimerkkikoodi 4:ssä näkyvä NewTaskRelatedField-muuttujan arvoon on lisätty tarkentavia tietoja. Tällä tavalla päästään käsiksi HTML-komponentin sisäisiin komponentteihin.

4.3 Testitapausten toteutus

Seuraavaksi oli aika tehdä ensimmäinen testitapaus, joka luonnollisesti oli web-sovellukseen sisäänkirjautumisen validointi. Vaikka sisäänkirjautuminen itsessään on oma testitapauksensa, oli tulevien testitapausten kannalta järkevämpää tehdä sisäänkirjautumisesta oma avainsana. Tällä tavalla sisäänkirjautuminen saataisiin helposti osaksi muitakin testitapauksia, koska jokainen testitapaus alkaa aina web-sovellukseen sisäänkirjautumisella. Esimerkkikoodi 4:ssä näkyy itse tehdyn avainsanan toteutus.

```
Login to Greip App
     [Documentation]         Login to the web application with given
                             username, password.
     [Arguments]             ${username}    ${password}
     Fill text               id=email       ${username}
     Fill secret             id=password    $password
     Click                   id=next
     Wait for navigation     ${APP_URL}     wait_until=networkidle
```

Esimerkkikoodi 5. Avainsana web-sovelluksen sisäänkirjautumiselle.

Kuten luvussa 3.4 kerrottiin, testitapaukset jakautuvat pääsääntöisesti kahteen kategoriaan, jotka olivat tietojen tarkastelu ja tietojen lisääminen. Sisäänkirjautumisen validoimisen jälkeen siirrytään testitapauksiin, jotka liittyvät tietojen tarkasteluun. Näiden testitapausten pohjimmainen tarkoitus on varmistaa, että web-sovelluksessa näkyy oikea arvo oikeassa paikassa. Web-sovelluksessa on erilaisia sivuja, joissa näytetään immateriaalioikeuksiin liittyvää dataa numeroina, päivämäärinä tai tekstinä. Tämänlaista dataa on esimerkiksi erilaisten patenttien määrä, jonkin patentin tärkeitä määräaikoja tai patentin rekisteröintinumero.

Tässä kohtaa huomattiin, ettei ole järkevää kovakoodata näiden testitapausten validoimiseen tarvittavaa dataa suoraan testitapauksiin. Jos testitapaukset olisi tehty näin, olisi niiden ylläpito ollut tulevaisuudessa työlästä ja päätiedoston koko olisi taas kasvanut turhaan. Näiden ongelmien ratkaisemiseksi kaikki näiden testitapausten validoimiseen tarvittava data siirrettiin omaan tiedostoon. Tämä tiedosto lisättiin päätiedostoon asetusosiossa ryhmäksi muuttujia, joita pystyi sitten käyttämään testitapauksissa. Samalla oli syytä miettiä, minkälainen tapa toteuttaa validointi olisi kaikista tehokkain. Testitapauksille, joiden tarkoitus oli validoida, että data näkyy käyttöliittymässä oikein, luotiin omat avainsanat. Nämä avainsanat käyttivät validointitiedoston dataan hyödyksi suorittaessaan validointia. Esimerkkikoodi 5:ssä nähdään yksi näistä avainsanoista.

```
Check case page information
    [Documentation]  Checks that all information is correct in a case
landing page
    [Arguments]  ${testdata}  ${account}=None  ${login}=True
    IF  ${login} == True
    ${account}=  Parse account variable  ${account}
    Login to Greip App  ${account}[0]  ${account}[1]
    END
    Go to  ${APP_URL}case/${testdata}[CaseType]?caseId=${testdata}[CaseId]
    FOR  ${key}  IN  @{testdata}
        IF  "${key}" == "CaseType"
        Log  Skipped: ${key}
        ELSE IF  "${key}" == "CaseId"
        Log  Skipped: ${key}
        ELSE
        Check value in element  data-test-id=${key}  ${testdata}[${key}]
        END
    END
```

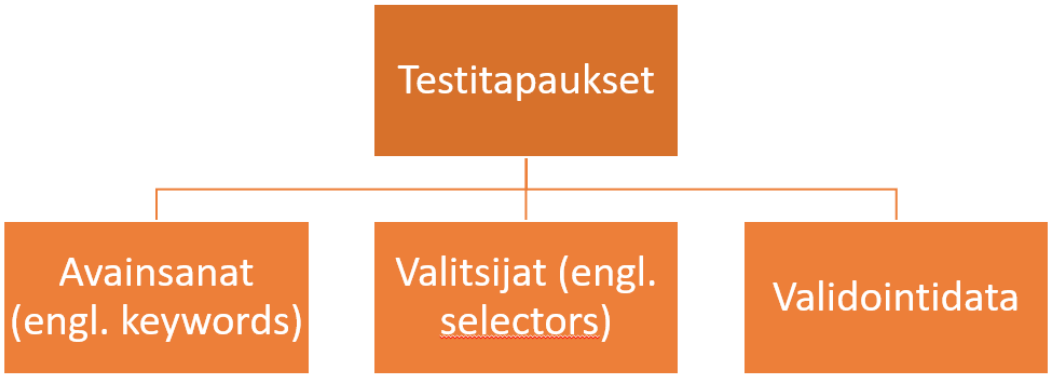
Esimerkkikoodi 6. Avainsana suojamuotojen tietojen validoinnille.

Kun kaikki tietojentarkasteluun liittyvät testitapaukset olivat tehty, oli vuorossa tietojen lisäämiseen liittyvät testitapaukset. Nämä sisälsivät kaikki testitapaukset, joissa käyttäjä tekee muutoksia tai lisäyksiä dataan, jotka näkyvät jonkinlaisina muutoksina käyttöliittymässä. Näissä testitapauksissa keskityttiin siihen, että toiminnot, joita käyttäjät useimmiten käyttävät käyttöliittymässä, toimivat oikein. Testitapaukset toteutettiin hyvin perinteellisellä tavalla laittamalla Browser-kirjasto suorittamaan toiminnallisuus vaihe vaiheelta. Sen jälkeen aina validoitiin, että odotettu muutos on tapahtunut käyttöliittymässä. Esimerkkikoodi 7:ssä näkyy yksi tällainen testitapaus, jossa lisätään kommentti käyttöliittymässä työtilaan, jonka jälkeen se validoidaan tarkistamalla, että syötetty kommentti löytyy käyttöliittymästä.

```
Add comment to Task
    [Documentation]  Adds a comment to case task and validates
    [Tags]  AddTaskComment  Task
    Login to Greip App  ${AC1_USER}  ${AC1_USER_PW}
    Go to  ${APP_URL}duedate/casetask?activityItemId=5
    Click  ${Editor}  position_x=2  position_y=2
    ${cur_time}=  Get current date
    Keyboard Input  insertText  This is a comment ${cur_time}
    Click  data-test-id=Submit
    Check value in element  ${Discussion}  This is a comment ${cur_time}
```

Esimerkkikoodi 7. Testitapaus kommentin lisääminen työtilaan käyttöliittymässä.

Joidenkin testitapausten validointi oli haastavaa, koska käyttöliittymässä tapahtuvat muutokset eivät olleet helposti havaittavissa Browser-kirjastolla. Nämä muutokset olivat enemmän visuaalisia muutoksia, kuten värien vaihtumista toiseen. Näillä värien vaihtumisilla kuvataan erilaisten tilojen muutosta käyttöliittymässä. Tämän ongelman ratkaisemiseksi lisättiin HTML-elementtiin, jossa muutos näkyi, dynaaminen data-test-id -attribuutti. Tähän attribuuttiin otettiin mukaan arvo, joka kertoi, mikä tila oli kyseessä. Testitapaukset pystyttiin tämän avulla validoimaan helposti, kun tila pystyttiin tarkistamaan suoraan attribuutista. Kuvassa 5 näkyy testauksessa tarpeellisten tiedostojen lopullinen rakenne.



Kuva 5. Testauksen tiedostorakenne

Tämän rakenteen avulla testaukseen tarvittava data oli hajautettu siten, että muutoksia dataan pystyi tekemään helposti ja oli kokonaisuutena helpommin hallinnoitavissa.

Lopulta testit pystyttiin ajamaan komentoriviltä annetulla komennolla. Testiajon päätteeksi Robot Framework loi testien läpikäynnistä raportin, jossa näkyi, olivatko testit onnistuneet vai eivät. Mahdolliset virheet testeissä korjattiin, jotta kaikki testit antoivat testitapauksissa määritellyn lopputuloksen. Kaikki testaukseen tarvittavat neljä tiedostoa vietiin web-sovelluksen koodikantaan, jotta niitä pystyttäisiin käyttämään putken toteutuksessa.

4.4 Putken toteutus

Kun testitapaukset olivat valmiit, oli vuorossa testaukselle oman putken kehittäminen, jotta se saataisiin osaksi kehitysprosessia ja jotta testitulosten raportti olisi helposti saatavissa. Koska yrityksessä on jo käytössä Azure Devops ja siellä erilaisia putkia, oli luontevaa toteuttaa automaatiotestaus omaksi putkeksi. Tässä vaiheessa web-sovellus oli käynnissä Azure-pilvipalvelussa testaukseen tarkoitetussa ympäristössä.

Putken toteutus aloitettiin käyttämällä Azure DevOps -palvelusta löytyvää Pipelines-osiota, jolla putken alustaminen onnistui helposti. Kun uutta putkea alettiin luomaan, täytyi valita missä web-sovelluksen koodi sijaitsee. Tähän kohtaan valittiin Azure Repos Git, koska yrityksen kaikki koodi sijaitsee Azureen integroidussa Git-versionhallintaohjelmassa. Seuraavaksi valittiin web-sovelluksen repositorio (engl. repository). Tämän jälkeen täytyi valita pohja putkelle. Tähän valittiin Python-paketteja käsittelevä pohja, koska Robot Framework on toteutettu Python-ohjelmointikielellä ja tähän pohjaan tuli valmiina tarpeellisia toimia, joita tarvittiin testauksen ajossa.

Työkalu loi YAML-tiedoston, joka sisälsi tarvittavat toiminnot, jolla pystyy esimerkiksi testaamaan Python-ohjelmia. Pohjassa oli määritelty, että putki ajettaisiin automaattisesti aina, kun web-sovelluksen pääsäiliöön tehtäisiin muutoksia. Tässä vaiheessa ei haluttu, että putkella on mitään laukaisijaa (engl. trigger), joten se määriteltiin pois päältä. Tässä pohjassa oli turhia kohtia kuten putken askelten (engl. step) läpikäyminen monella eri Python-versiolla. Robot Framework -testien läpikäyminen eri Python-versioilla ei olisi hyödyllistä, joten tästä karsittiin pois ylimääräiset versiot ja jätettiin vain 3.7-versio. Myös virtuaalikone, joka ajaa putken toiminnot, asetettiin tässä vaiheessa, ja se määriteltiin käyttämään viimeisintä Windows-käyttöjärjestelmää. Esimerkkikoodi 8:ssa näkyy, miltä nämä asetukset näyttivät YAML-tiedostossa.

```
trigger:
  - none

pool:
  vmImage: "windows-2019"
strategy:
  matrix:
    Python37:
      python.version: "3.7"
```

Esimerkkikoodi 8. Ote putken YAML-määrittelytiedostosta.

Seuraavaksi täytyi määritellä ensimmäinen putken askel. Tässä askeleessa otettiin käyttöön esimerkkikoodi 8:ssa näkyvästä strategiamatriisissa oleva Python-versio, jota käytetään seuraavissa putken askeleissa.

Jotta virtuaalikone pystyy ajamaan Robot Framework -testejä, on siinä oltava kaikki testien ajossa tarvittavat ohjelmistot. Virtuaalikoneen mukana tulee automaattisesti jo paljon tarpeellisia ohjelmistoja, kuten Node.js. Seuraavassa askeleessa asennettiin virtuaalikoneeseen kaikki muut tarpeelliset ohjelmistot, kuten Robot Framework ja Browser-kirjasto. Tämä toteutettiin antamalla askeleessa virtuaalikoneelle komentoja sen komentoriville. Myös Browser-kirjasto alustettiin omalla komennolla, jotta se lataisi web-selainmoottorit virtuaalikoneelle.

Seuraava askel on ajaa itse testit virtuaalikoneella. Virtuaalikoneelle ladataan aina putken käynnistyessä sille asetetusta Git-repositorion haarasta web-sovelluksen koodikanta. Koska kaikki testauksessa tarvittavat tiedostot löytyvät koodikannasta, on virtuaalikoneella pääsy näihin tiedostoihin. Tässä askeleessa haluttiin myös käyttää putkimuuttujia (engl. pipeline variables). Ei olisi järkevää jättää esimerkiksi sisäänkirjautumiseen tarvittavia käyttäjätietoja koodikantaan, joten nämä tiedot poistettiin testitiedostosta. Nämä tiedot annettiin sen sijaan salaisina putkimuuttujina Robot Frameworkin testiympäristön sisälle suoraan komentorivikomennossa, jolla testiajo käynnistetään.

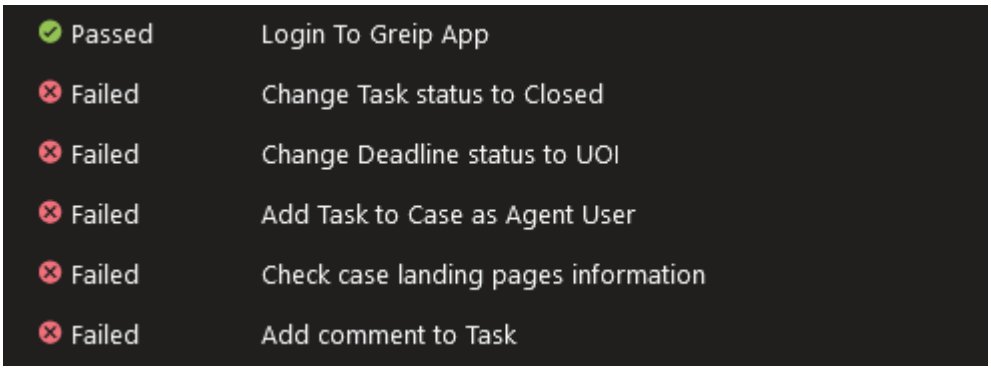
Muuttujia määriteltiin kaiken kaikkiaan käyttäjätunnus, käyttäjätunnuksen salasana ja URL-osoite, jossa testattava web-sovellus on. Komentoon lisättiin myös parametri, jonka avulla testiajosta muodostuva raportti luotaisiin myös XML-muodossa. Esimerkkikoodi 9:ssä näkyy, miltä testejä ajava askel lopulta näytti YAML-tiedostossa.

```
# Runs Robot Framework tests
  - powershell: |
      robot -x outputxunit.xml -v AC1_USER:$(TestUser) -v
AC1_USER_PW:$(TestUserPassword) -v APP_URL:$(BaseUrl)
"$(Build.SourcesDirectory)\tests\Functional\Automated UI testing\UI-
tests.robot"
    displayName: "Run Robot Test"
```

Esimerkkikoodi 9. Testejä ajava askel YAML-määrittelytiedostossa.

Viimeinen askel on testiajosta luodun raportin julkaisu Azure DevOps -palveluun. Testitulosten julkaisua varten on olemassa oma tehtävä, jota käyttämällä tämä onnistui helposti. Aikaisemmassa askeleessa määriteltiin Robot Framework luomaan raportista XML-tiedosto. Tämä XML-tiedoston formaatti on yhteensopiva

Azure DevOpsin kanssa. Tehtävä määriteltiin hakemaan tämä XML-tiedosto kansiosta, johon se oli luotu virtuaalikoneessa. Tärkeää oli asettaa tämän tehtävän ehdoksi, että tehtävä suoritetaan, vaikka testien ajo epäonnistuisi. Jos yksikin testi epäonnistuu, olettaa putki, että koko putken toiminta on pysäytettävä ja tämän takia ehdon asettaminen on tärkeää. Kuvassa 6 näkyy ote, miltä testien raportti näyttää Azure DevOps -palvelussa.



Kuva 6. Ote testiraportista Azure DevOps -palvelussa.

Putki valmistui käytettäväksi, mutta monet testeistä epäonnistuivat, kuten kuvassa 6 näkyy. Tämä johtui siitä, että tietokanta, johon testattava web-sovellus oli yhdistetty, ei ollut alustettu testeihin tehdyllä testidatalla. Tähän ongelmaan ja muihin automaatiotestauksen puutteisiin ja parannuksiin paneudutaan seuraavassa luvussa.

5 Työn arviointi ja jatkokehitys

Tässä luvussa arvioidaan tehtyä käyttöliittymän automaatiotestauksen toteutusta ja käydään läpi mahdollisia jatkokehitysideoita.

5.1 Työn arviointi

Työn tavoitteet täyttyivät pääsääntöisesti hyvin. Automaatiotestauksesta saatiin helposti lähestyttävä ja ymmärrettävä esimerkiksi erottamalla monimutkaisemmat avainsanat omaan tiedostoonsa. Samaa lähestymistapaa käytettiin myös valitsijoiden ja validointiin tarvittavien tietojen kanssa. Tämä teki testauksen rakenteen helpommaksi hallita.

Käyttöliittymään tehty muutos, data-test-id -attribuuttien lisääminen, on varmasti paras mahdollinen tapa toteuttaa elementtien valitsemisen käyttöliittymästä. Kun ymmärrys testien toiminnallisuudesta kasvaa yrityksessä, on kehittäjien helpompi huomata, mitkä elementit ovat tarpeellisia automaatiotesteissä ja he osaavat varoa, etteivät testit mene turhaan rikki, kun muutoksia käyttöliittymään tehdään. Toisaalta, käytetty tapa minimoi mahdolliset tilanteet, missä testit voisivat mennä rikki verrattuna muihin tapoihin.

Putken toiminta on suurimmalta osin valmis ja siitä saadaan tuotua testiajosta muodostuva raportti Azure DevOpsiin. Putkeen on lisättävä vielä muutama toiminto, kun uusi ympäristö on pystytetty, jossa tietokantaa voi alustaa vapaasti. Näiden toimintojen lisääminen on suhteellisen triviaalia, eikä sen pitäisi tuoda paljoakaan lisätyötä, jotta testauksesta saisi todellisesti automatisoidun ja että yöllisistä sovelluksen koonneista saadaan kattavaa testaustietoa.

5.2 Jatkokehitys

Vaikka automaatiotestausprojekti oli suurimmilta osin valmis, jäi monta asiaa uupumaan toteutuksesta, joita käydään läpi tässä luvussa.

Testit on tehty niin, että tietokannan data oletetaan olevan aina tietyssä tilassa ennen testien ajoa. Jotta tämä olisi putkessa mahdollista, olisi tietokanta aina alustettava datalla, joka olisi tässä halutussa tilassa. Tämä voitaisiin toteuttaa putkessa siten, että virtuaalikone muodostaa yhteyden SQL-tietokantaan ja alustaa sen halutulla datalla aina ennen testien ajoa. Alustettavan datan voisi lisätä koodikantaan SQL-tiedostona, jolloin virtuaalikoneella olisi data käytettävissä putkessa. Olisi myös hyvä, että luotaisiin testattavalle web-sovellukselle oma ympäristö, joka olisi yhdistetty tietokantaan, joka voitaisiin alustaa huoletta aina ennen testien ajoa.

Monet testitapaukset, joita ei tähän projektiin otettu mukaan, liittyvät päivämääriin. Käyttöliittymässä tapahtuu muutoksia esimerkiksi silloin, kun johonkin määräaikaan on tietyn verran päiviä jäljellä. Testidataa on muokattava aina ennen tietokannan alustusta, jotta päivämäärät testidatassa olisivat aina tietyn monta päivää tulevaisuudessa. Tätä varten koodikantaan voisi lisätä skriptin, mikä pystyisi muokkaamaan tietokannan alustavan SQL-tiedoston siten, että halutut päivämäärät olisivat tietyn verran tulevaisuudessa testauksen ajankohdasta. Putkeen voisi lisätä askeleen, jossa skripti ajettaisi ja tietokanta alustettaisiin sen jälkeen muokatulla SQL-tiedostolla.

Ominaisuus, jota ei nyt tähän projektiin vielä lisätty mukaan, on Robot Frameworkin tunnisteet (engl. tags). Kaikille testitapauksille on mahdollista antaa yksi tai useampi tunniste. Näiden tunnisteiden avulla pystyy ajamaan vain tiettyjä testejä, jos esimerkiksi haluttaisiin testata nopeasti vain jokin tietty ominaisuus käyttöliittymästä. Tunnisteet, jotka halutaan ajaa, täytyy lisätä komentorivikomentoon, kun testiajo käynnistetään. Tämän voisi toteuttaa putkessa käyttäen putkimuuttujaa. YAML-tiedostoon pitäisi luultavasti lisätä ehtoja, joiden avulla määritellään, käytetäänkö tunnisteita testiajossa vai ei.

Kaikki testit ajettiin vain yhdellä web-selainmoottorilla, mutta Browser-kirjasto antaa mahdollisuuden käyttää muitakin moottoreita. Olisi hyödyllistä, että kaikki testit ajettaisiin kaikilla mahdollisilla moottoreilla, jotta testeistä saataisiin enemmän tietoa irti web-sovelluksen toimivuudesta. Tämän voisi toteuttaa putkessa

siten, että testit käydään erikseen läpi kaikilla moottoreilla ja näistä ajoista julkaistaan omat raportit Azure DevOpsiin.

Pienellä vaivalla tämän projektin pohjalta voisi toteuttaa myös rajapinnan automaatiotestauksen. Suositusta Pythonille tehdystä Requests-kirjastosta [23], jolla tehdään HTTP-kutsuja rajapintoihin, on olemassa myös Robot Framework yhteensopiva versio. [24.] Käyttöliittymän testaukseen tehtyä putkea pystyisi käyttämään myös rajapinnan testaukseen pienellä lisätyöllä.

6 Yhteenveto

Insinöörityön alussa tutustuttiin ohjelmistotestaukseen yleisellä tasolla, jossa käytiin läpi testauksen merkitystä, yleisimmät lähestymistavat ja testauksen tasot. Myös automaatiotestaukseen perehdyttiin ja sen vaikutuksesta ohjelmistokehitysprosessiin katselmoitiin. Insinöörityön projektina toteutettiin automaatiotestaus Greip eService-web-sovellukselle ja tämän automaatiotestauksen lisääminen omaksi putkeksi. Tämä prosessi käytiin vaihe vaiheelta läpi. Näihin vaiheisiin kuului toteutuksen kannalta kaikki tarpeelliset työvaiheet, kuten muun muassa testitapausten määrittely, tarpeellisten käyttöliittymämuutosten toteutus ja testidatan luonti. Myös testattavaan käyttöliittymään perehdyttiin ja käytetyt teknologiat alustettiin. Lopuksi projektia arvioitiin sen tavoitteiden näkökulmasta.

Projekti sujui suurimmilta osin sujuvasti. Päätös toteuttaa automaatiotestaus Robot Framework:in avulla osoittautui hyväksi valinnaksi. Robot Frameworkin dokumentaatio on kattava ja aina ongelmia kohdatessa sieltä löytyi hyvin tietoa ongelman ratkaisuun. Myös Browser-kirjaston dokumentaatio oli yhtä laadukasta kuin Robot Frameworkin. Se, ettei täytynyt itse alkaa suunnittelemaan testitapauksia, nopeutti projektin etenemistä huomattavasti. Ihmetystä aiheutti se, ettei Azure:ssa ollut putkille mitään valmista pohjaa Robot Framework -testeille. Vaikka valmista pohjaa ei suoraan ollut tällaisen tarkoitukseen, se ei tuottanut kauheasti ongelmia. Onneksi putkien tekoon löytyi paljon hyviä ohjeita internetistä.

Lopulta voidaan todeta, että insinöörityön tavoitteet saavutettiin hyvin. Toteutuksesta tuli kokonaisuutena helposti ymmärrettävä ja lähestyttävä. Jatkokehitykselle on valtavasti mahdollisuuksia ja tämän työn pohjalta niiden toteuttaminen onnistuu varmasti sujuvasti. Kun automaatiotestaus saadaan osaksi kehitysprosessia, on siitä selvästi hyötyä yritykselle varmistamalla sovelluksen toiminnallisuuden päivittäisten koontien yhteydessä ja varmistamalla, että mahdolliset regressio-ongelmat havaitaan nopeammin kuin pelkällä manuaalisella testauksella.

Lähteet

1 Kelion, Leo. 2015. Airbus A400M plane crash linked to software fault. Verkkoaineisto. BBC. <https://www.bbc.com/news/technology-32810273>. Luettu 14.10.2021.

2 Rajkumar. 2021. What Is Software Testing | Everything You Should Know. Verkkoaineisto. Software Testing Material. <https://www.softwaretesting-material.com/software-testing/>. Luettu 14.10.2021.

3 Kausarinen, Jussi Pekka. 2017. Ohjelmistotestauksen käsikirja. E-kirja.

4 4 Levels of Software Testing: Performers, Steps, and Objectives. 2020. Verkkoaineisto. UTOR. <https://u-tor.com/topic/software-testing-levels>. Luettu 14.10.2021.

5 Gill, Navdeep Singh. 2021. Unit Testing Techniques and Best Practices. Verkkoaineisto. Xenonstack. <https://www.xenonstack.com/insights/what-is-unit-testing>. Luettu 14.10.2021.

6 Byk, Viktoria. 2015. Levels of Automation in Software Testing. Verkkoaineisto. QATestLab. Päivitetty 2020. <https://blog.qatest-lab.com/2015/08/19/test-automation-levels/>. Luettu 11.03.2022.

7 Mihailescu, Clément. 2020. All You Need To Know About TypeScript. Verkkoaineisto. <https://www.youtube.com/watch?v=eCZhz0JCVx0>. Kuunneltu 20.01.2022.

8 Hamedani, Mosh. 2018. What Is React (React js) & Why Is It So Popular? Verkkoaineisto. <https://www.youtube.com/watch?v=N3AkSS5hXMA>. Kuunneltu 20.01.2022.

9 React. Meta Platforms, Inc. Verkkoaineisto. <https://reactjs.org/>. Luettu 20.01.2022.

10 Material UI. Material-UI SAS. Verkkoaineisto. <https://mui.com/>. Luettu 21.01.2022.

11 Robot Framework. Robot Framework ry. Verkkoaineisto. <https://robotframework.org/>. Luettu 27.01.2022.

12 Playwright: A New Test Automation Framework for the Modern Web [Webinar Recording]. 2021. Verkkoaineisto. Applitools. <https://www.youtube.com/watch?v=_Jla6DyuEu4]>. Kuunneltu 28.01.2022.

13 Browser Library. Verkkoaineisto. <https://robotframework-browser.org/>. Luettu 28.01.2022.

14 Browser-kirjaston avainsanadokumentaatio. 2022. Verkkoaineisto. <https://marketsquare.github.io/robotframework-browser/Browser.html>. Luettu 28.01.2022.

15 Playwright. Microsoft. Verkkoaineisto. <https://playwright.dev/>. Luettu 28.01.2022.

16 Node.js. OpenJS Foundation. Verkkoaineisto. <https://nodejs.org/en/>. Luettu 28.01.2022.

17 Azure DevOps. Microsoft. Verkkoaineisto. <https://azure.microsoft.com/en-us/services/devops/#overview>. Luettu 17.02.2022.

18 What is Azure Pipelines? 2022. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>. Luettu 17.02.2022.

19 CICD. Synopsys, Inc. Verkkoaineisto. <https://www.synopsys.com/glossary/what-is-cicd.html>. Luettu 17.02.2022.

20 What is DevOps? Amazon Web Services, Inc. Verkkoaineisto. <https://aws.amazon.com/devops/what-is-devops/>. Luettu 17.02.2022.

21 Use Azure Pipelines. 2022. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/pipelines-get-started?view=azure-devops>. Luettu 17.02.2022.

22 Using data attributes. MDN Web Docs. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes>. Luettu 28.01.2022.

23 Requests-kirjasto. Verkkoaineisto. <https://docs.python-requests.org/en/latest/>. Luettu 10.03.2022.

24 Robot Framework Requests. Verkkoaineisto. <https://marketsquare.github.io/robotframework-requests/doc/RequestsLibrary.html>. Luettu 10.03.2022.