Browser libraryn käyttöönotto Robot Frameworkilla ja SeleniumLibrary-testien korvaaminen

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutus
syksy 2023

Nico Polvi

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutus Tiivistelmä

Tekijä Nico Polvi Vuosi 2023

Työn nimi Browser libraryn käyttöönotto Robot Frameworkilla ja SeleniumLibrary-testien korvaaminen

Ohjaaja Esa Huiskonen (HAMK), Timo Ihalempiä (Trimble)

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli ottaa käyttöön web-testauskirjasto Browser library Robot Framework -automaatiokehyksellä sekä korvata nykyisiä SeleniumLibrary-kirjastoa käyttäviä testisarjoja Browser libraryn avainsanoilla. Työllä pyrittiin selvittämään, onko kirjaston vaihtamisella hyötyä testauksen kannalta. Käytännön osuudessa tarjotaan asennusopas testiympäristön ja kirjaston käyttöönotolle, sekä kerrotaan testien korvaamisprosessista. Testien korvaamisen jälkeen vertailtiin testien suoritusaikoja sekä luotettavuutta kirjastojen välillä. Opinnäytetyön toimeksiantajana oli Trimble Solutions.

Tämän toiminnallisen opinnäytetyön teoriaosuudessa kerrotaan ensin ohjelmistotestauksesta, jonka jälkeen keskitytään testiautomaatioon. Lisäksi esitellään web-testauksen menetelmiä, joiden varaan käytännön osuudessa käytettävät testisarjat pohjautuvat. Ennen käytännön osuuteen siirtymistä esitellään vielä työn toteutustapa ja työhön liittyvät teknologiat.

Opinnäytetyön lopputuloksena havaittiin Browser library -kirjastoa hyödyntävien testitapausten olevan suoritusajoiltaan nopeampia kuin käytössä olevien SeleniumLibrary-testitapausten. Browser libraryn asennus oli suoraviivaista ja sen ylläpito sekä päivitykset vaikuttavat helpommalta prosessilta kuin SeleniumLibraryn. Tulosten sekä käyttökokemuksen perusteella Browser library -kirjastoon siirtymiselle on vahvat perusteet. Varsinkin uusien hankkeiden kohdalla Browser libraryn käyttöä kannattaa harkita.

Avainsanat Testiautomaatio, Robot Framework, SeleniumLibrary, Browser library

Sivut 57 sivua ja liitteitä 15 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology — Abstract

Author — Nico Polvi — Year 2023

Subject — Initialization of Browser library in Robot Framework and migration of tests from SeleniumLibrary to Browser library

Supervisor — Esa Huiskonen (HAMK), Timo Ihalempiä (Trimble)

ABSTRACT

The purpose of this thesis was to take a new web testing library called Browser library into use and replace some of the current test cases, using the SeleniumLibrary, with Browser library's keywords. The aim was to discover if changing the library would be beneficial for testing. The practical part includes a guide of installing the testing environment and library itself. Moreover, the practical part includes documentation about the replacement process and short comparison of execution time and reliability between the test cases. The commissioner of the thesis was Trimble Solutions.

The thesis was carried out as a practical project. The theoretical part starts with introducing software testing and test automation. Additionally, different web testing methods, to which test cases in practical part are based on, are explained. Before moving to the practical part, the project method and technologies related to this thesis are introduced.

The result of the thesis was that test cases using Browser library are faster than test cases using SeleniumLibrary. The installation of Browser library was a straightforward process. The maintenance and updating of Browser library seem to be easier than in SeleniumLibrary. Based on the test results and usage experience, it can be argued that it would be reasonable to migrate from SeleniumLibrary to Browser library. Especially when initiating new projects, Browser library could be recommended.

Keywords — Test automation, Robot Framework, SeleniumLibrary, Browser library

Pages — 57 pages and appendices 15 pages

Sanasto

Framework	Kehys tai runko ohjelmistolle, jonka päälle varsinainen toiminnallisuus rakennetaan
http	Hypertext Transfer Protocol, tiedonsiirtoprotokolla
WebSocket	tiedonsiirtoprotokolla
I/O	Input/Output, tiedon siirtäminen laitteistokomponenttien tai sovelluskomponenttien välillä
Selector	Browser libraryn käyttämä termi/määritelmä elementin paikantamiselle sivustolta
Locator	SeleniumLibraryn käyttämä termi/määritelmä elementin paikantamiselle sivustolta
pip	Pythonin paketinhallintajärjestelmä, jolla voidaan ladata ja asentaa ohjelmia
metadata	tieto, jolla kuvataan toista tietoa
binääritiedosto	tietokoneen luettavissa oleva, mikä tahansa tieto
debuggaus	Ohjelmistosta havaitun virheen paikallistaminen
xpath	ohjelmointikieli, jolla voidaan tunnistaa ja paikallistaa elementtejä XML-tiedoston sisällä
LTS	Long-term support, ohjelmistojulkaisijan lupaama pitkäaikainen tuki kyseiselle ohjelmiston versiolle
backend	sivuston tai sovelluksen palvelinpuoli, jossa varmistetaan tiedonsiirron toimivuus
frontend	sivuston tai sovelluksen selainpuoli, joka tarkoittaa käyttäjälle näkyvää osaa
I/O	Input/Output, tiedonsiirrántään liittyvät termit. Input tarkoittaa laitteelle tai sovellukselle annettavaa syötettä. Output on tuloste, joka saadaan Input-syötteen käsittelyn tuloksena.
CSS	Cascading Style Sheets, verkkosivujen ulkoasun tyylikieli

Sisällys

1 Johdanto ... 1

2 Ohjelmistotestaus ... 3

2.1 Ohjelmistotestauksen määritelmä ... 3

2.2 Testaus osana sovelluskehitystä ... 4

2.3 Testaustasot ... 6

2.3.1 Yksikkötestaus ... 6

2.3.2 Integrointitestaus ... 7

2.3.3 Järjestelmätestaus ... 7

2.3.4 Hyväksymistestaus ... 8

2.4 Testiautomaatio ... 8

2.5 Testiautomaation hyödyt ja haasteet ... 9

3 Testausmenetelmät ... 11

3.1 Web-testaus ... 11

3.2 Regressiotestaus ... 14

3.3 Päästä päähän -testaus ... 15

3.4 Savutestaus ... 17

4 Kehittämistyön menetelmät ja teknologiat ... 18

4.1 Selenium ... 20

4.2 Playwright ... 22

4.3 Node.js ... 23

4.4 Robot Framework ... 23

4.4.1 SeleniumLibrary ... 25

4.4.2 Browser library ... 26

4.5 Robot Frameworkin testidatan rakenne ja kirjoitussyntaksi ... 27

5 Ohjelmistojen asennus ja käyttöönotto ... 30

5.1 Pythonin ja Robot Frameworkin asennus ... 30

5.2 Node.js ... 31

5.3 Browser libraryn asennus ... 31

5.4 Kehitysympäristöjen asennus ... 32

6 Testien korvaaminen ... 33

6.1 Avainsanojen korvaaminen suoraan testeihin ... 33

6.2 Testien korvaaminen käyttäen omaa resurssitiedostoa ... 40

6.3 Testien suoritusaikojen vertailu kirjastojen välillä ... 46

7 Johtopäätökset ja pohdinta 50
8 Yhteenveto 53
Lähteet 54

Kuvat, ohjelmakoodit, komennot ja taulukot

Komento 1 Pythonin versionumeron sekä asennuksen onnistumisen tarkastuskomento30
Komento 2 robotframework-browser-migration asennus........42

Kuva 1 Sovelluksesta löydetyn virheen kustannusarvio sovelluskehityksen eri vaiheissa (mukaillen Cavero-Baptista, n.d.) 12
Kuva 2 Web-testauksen tasot testaustasoja mukaillen (mukaillen Knight, 2022)........17
Kuva 3 Kanban-taulu Microsoftin Plannerissa........19
Kuva 4 Teknologiat Robot Frameworkin ja selainautomaation taustalla 20
Kuva 5 Kuvaus Seleniumin arkkitehtuurista yhteyden muodostamisen osalta (mukaillen Lambdatest, n.d.)........21
Kuva 6 Kuvaus Playwrightin arkkitehtuurista yhteyden muodostamisen osalta (mukaillen Lambdatest, n.d.)........23
Kuva 7 Robot Frameworkin arkkitehtuuri yleisellä tasolla (mukaillen Robot Framework, n.d.b.)........25
Kuva 8 Testidatan rakenne kansiopolussa 34
Kuva 9 Tekla.com-testidatan rakenne kansiopolussa 40
Kuva 10 SeleniumLibrary-avainsanojen statistiikka 43

Ohjelmakoodi 1 Testidatan käsittelyn syntaksi Space Separated Format -tyyliä käyttäen29
Ohjelmakoodi 2 Selaimen käynnistämisen kustomoitu avainsana 36
Ohjelmakoodi 3 Sisäänkirjautumistietojen syöttäminen SeleniumLibrary-avainsanoja käyttäen vs. Browser library -avainsanoja käyttäen 37
Ohjelmakoodi 4 Erot kirjastojen toteutuksissa liittyen elementtien tunnistamiseen sivustolla 39
Ohjelmakoodi 5 Raporttinäkymä sekä korvaavan avainsanan toteutus resurssitiedostossa 41

Ohjelmakoodi 6 Kirjaston määritys tiedoston sisällä 44
Ohjelmakoodi 7 Selaimen vaihto SeleniumLibrary-testeissä sekä Contextin vaihto Browser library-testeissä 45
Ohjelmakoodi 8 Resurssitiedoston määritys tiedoston sisällä avainsanojen käyttöönottoon 46

Taulukko 1 Testitiedoston rakenne otsikon ja käyttötarkoituksen mukaan (Robot Framework, n.d.b.)..27
Taulukko 2 Erot testien suoritusajoissa kirjastojen välillä (sekunneissa)........................49

Liitteet

Liite 1 Aineistonhallintasuunnitelma
Liite 2 Asennusdokumentaatio
Liite 3 Testeissä käytetyt SeleniumLibrary-avainsanat ja niiden vastineet Browser librarystä

1 Johdanto

Testaus on tärkeä sovelluskehityksen laadunvarmistuksen osa-alue. Testauksella varmistetaan, että sovellus toimii määrittelyjen mukaisesti ja vastaa asiakkaan liiketoimintavaatimuksia. Testausta voidaan suorittaa joko manuaalisesti tai automaation avulla. Tässä opinnäytetyössä keskitytään verkkoselainpohjaisten sovellusten käyttöliittymän automaatiotestaukseen.

Opinnäytetyön teoriaosuuden tavoitteena on kertoa testauksesta ja erityisesti testiautomaatiosta yleisellä tasolla. Menetelmien osuudessa kuvataan ensiksi työn toteutustapa, minkä jälkeen kerrotaan käytännön osuudessa käytettävistä teknologioista ja ohjelmistoista. Käytännön osuus lähtee liikkeelle asennusdokumentaatiosta. Dokumentaatiossa pyritään kuvaamaan asennusvaiheet niin, että käyttäjä voi niiden perusteella asentaa testiympäristön omalle kotikoneelle. Asennusvaiheiden jälkeen raportoidaan työssä käytettävien testitapausten korvaamisesta käyttäen uuden testikirjaston avainsanoja.

Työn toimeksiantajana on teknologiateollisuusyritys Trimble Solutions. Suoritin työharjoittelun Quality Assurance -tiimissä, jonka tärkeimpänä vastuualueena on yrityksen kehityksessä olevien ohjelmistojen testaus. Tiimillä on ollut käytössä web-sovellusten testiautomaatiokehityksessä Robot Framework ja SeleniumLibrary. SeleniumLibrary on ollut pitkään yleisin valinta web-testauksen ja Robot Frameworkin käytön parissa, mutta vuonna 2020 julkaistu Browser library on noussut nopeasti haastamaan SeleniumLibraryn suosiota. Monissa yrityksissä on siirrytty kokonaan uuden Browser library -kirjaston käyttöön ja tämän opinnäytetyön tavoitteena onkin ottaa Browser library -kirjasto käyttöön sekä kokeilla useamman nykyisen SeleniumLibrary-testisarjan korvaamista Browser library -kirjastolla.

Työn tarkasteluun on valittu testisarjoja kahdesta eri projektista, joiden parissa olen työskennellyt työsuhteen aikana. Kokemukset ja havainnot, jotka testitapausten korvaamisen aikana tulevat eteen, toimivat pohdinnan välineenä työn lopussa. Työn tutkimuskysymykset ovat seuraavat:

- Miten testiautomaatiota voidaan hyödyntää web-sovelluksen kehityksessä?
- Mihin asioihin tulee kiinnittää huomiota, jos SeleniumLibrary-kirjastoon perustuvat testitapaukset halutaan lähteä korvaamaan Browser library -kirjastolla?
- Kannattaako kirjaston korvaamiseen lähteä tiimin nykyisissä projekteissa?

2 Ohjelmistotestaus

Ensimmäisessä teorialuvussa puhutaan yleisesti ohjelmiston testauksesta sekä sen roolista ohjelmistokehityksen parissa. Luvussa kerrotaan yleisesti testauksen eri tasoista. Tämän jälkeen keskitytään testiautomaation käsitteeseen ja ominaisuuksiin.

2.1 Ohjelmistotestauksen määritelmä

Ohjelmistotestaus on laadunvarmistustyötä, jolla varmistetaan ohjelmiston toimivuus määrittelyn ja suunnitelman mukaan. Testaus on jatkuvaa vertailua määrittelyjen ja ohjelman toiminnan välillä. Testauksen avulla pyritään löytämään mahdolliset ohjelman poikkeavuudet suunnitelmasta. (Kasurinen, 2014, Luku 1 Testaaminen yleisesti, Mitä on "ohjelmistotestaus")

Testaustyön tavoitteena on virheiden löytämisen lisäksi parantaa ohjelmiston tehokkuutta, tarkkuutta sekä käytettävyyttä (GeeksforGeeks, 2017). Hyvin tehdyn testaustyön avulla epäkohdat voidaan huomioida projektin alkuvaiheessa, mikä vahvistaa ohjelmiston luotettavuutta, tietoturvaa sekä suorituskykyä. Nämä puolestaan säästävät projektiin sidottua aikaa, parantavat tehokkuutta sekä lisäävät asiakastyytyväisyyttä (Guru, 2023 -a).

Ohjelmistotestaus on itsessään kokonaisuus, joka kuuluu laajempaan sovelluskehityksen kokonaisuuteen. Testaustyön sisältö voi olla hyvin moniulotteista kuten koodaamista, dokumentaation kirjoitusta tai haastattelujen pitämistä testattavan kohteen käyttäjille. Työn sisältö riippuu olennaisesti sovelluskehityksen työvaiheesta sekä testattavan kohteen ominaisuuksista. (Kasurinen, 2014, Luku 1 Testaaminen yleisesti, toinen kappale)

Ohjelmistotestaus koostuu eri osa-alueista. Kasurinen (2014, Luku 3 Testaamisen osa-alueet ja tasot, toinen kappale) mainitsee IEEE SWEBOK (Software Engineering Body of Knowledge) -mallin, jossa ammattimainen testaus jaetaan kuuteen eri pääkategoriaan. Pääkategoriat ovat testauksen perusasiat (mm. terminologia), testaustasot (testauksen kohde ja tavoitteet), testausmenetelmät, testauksen mittarit, testausprosessit sekä testaustyökalut. Edellämainitut kategoriat kattavat kaikki ne alueet, jotka testaustyötä tekevän henkilön tulisi hallita.

2.2 Testaus osana sovelluskehitystä

Ohjelmistotestaus on ohjelmistokehityksen laadunvarmistustyötä, jonka avulla varmistetaan käyttäjien tyytyväisyys ja luotettavuus sovelluksen käyttöä kohtaan. Usein ohjelmistokehitysprojekteissa yritetään säästää rahaa ottamalla resursseja pois testaustyöstä, jolloin toivotaan säästöjä aikataulun sekä budjetin suhteen. Yleensä tämä aiheuttaa kuitenkin juuri sen, että projektit venyvät ja ohjelmiston laatu kärsii. (Vala Group, n.d.a.)

Sovelluskehityksessä tulee aina vastaan ongelmia ja virheitä. Testaajat tuottavat tietoa kehitystiimille kehityksen aikana havaituista ongelmista sovelluksessa. Mitä aiemmin testaus otetaan osaksi projektia, sitä parempaa laatua sovelluksen käyttäjille pystytään tarjoamaan. (Vala Group, n.d.a.)

Testauksen rooli vaihtelee suuresti sovelluskehityksessä käytetyn mallin mukaan. Kasurinen (2014, Luku 3 Testaus erittäin lyhyesti) mainitsee yleisellä tasolla vaiheet, joiden mukaan testaus käytännössä etenee lähes jokaisen projektin kohdalla. Testaustyössä on osallisena kolme eri osapuolta: testaamisen vastuuhenkilö, sovelluskehittäjät sekä varsinaiset testaajat. Osapuolet laativat aluksi yhdessä testaussuunnitelman ja kirjoittavat myös ensimmäiset testitapaukset. Tämän jälkeen sovelluskehittäjät aloittavat toiminnallisuuksien kehittämisen ja suorittavat samalla niille yksittäisiä testejä.

Sitä mukaa kun toiminnallisuudet kehittyvät ja valmistuvat, testaajat ja kehittäjät päivittävät alussa luotua testaussunnitelmaa sekä lisäävät testaajien vastuulle uusia testitapauksia. Testaajien ilmoitusten perusteella kehittäjät korjaavat havaittuja virheitä sovelluksesta. Kun virheet ovat korjattu, elementit liitetään osaksi kehitettävää sovellusta ja varmistetaan niiden toimivuus yhdessä. (Kasurinen, 2014, Luku 3 Testaus erittäin lyhyesti)

Kun kaikki sovelluksen elementit ovat kehittäjien puolesta saatu valmiiksi ja niiden välisten integraatioiden testaus on saatu valmiiksi, alkaa koko sovellusta koskeva testausvaihe. Testaajat raportoivat jälleen eteen tulevia virheitä sovelluskehittäjille, jotka korjaavat niitä siihen asti, kunnes koko sovellus toimii määrittelyjen mukaisesti. (Kasurinen, 2014, Luku 3 Testaus erittäin lyhyesti)

Viimeiseksi ennen sovelluksen varsinaista käyttöönottoa varmistetaan sovelluksen toimivuus asiakkaan kanssa tehdyn vaatimusmäärittelyn mukaisesti. Tässä vaiheessa asiakas otetaan joko osaksi testausta tai asiakkaan edustaja suorittaa testauksen asiakkaan liiketoiminnan näkökulmasta. Tämän vaiheen jälkeen sovellus on valmis toimitettavaksi asiakkaan käyttöön. Valmiin projektin dokumentaatio arkistoidaan ja siihen voidaan palata, mikäli käyttöönoton jälkeen sovellukseen tarvitaan vielä muutoksia. (Kasurinen, 2014, Luku 3 Testaus erittäin lyhyesti)

Testauksen rooli projektissa vaihtelee sovelluskehityksessä käytetyn kehitysmallin mukaan. Mallit voidaan karkeasti jakaa kolmeen eri pääluokkaan, jotka ovat peräkkäisten vaiheiden malli, inkrementaalinen malli sekä iteratiivinen malli. (Homès, 2012, ss. 43–44)

Peräkkäisten vaiheiden mallien mukaisessa kehityksessä vaiheet etenevät järjestyksessä eteenpäin ja seuraavaan vaiheeseen siirrytään vasta edellisen vaiheen tultua valmiiksi (Homès, 2012, ss. 43–44). Yksi tunnetuimmista peräkkäisten vaiheiden mukaisesta kehitysmallista on vesiputousmalli. Tässä mallissa testaus tapahtuu projektin loppuvaiheilla, jolloin vaatimusmäärittely, suunnittelu ja toteutus on jo tehty kehityksen alkuvaiheessa. Tällöin testaustyössä keskitytään puhtaasti siihen, että ohjelmisto toimii määrittelyn ja suunnittelun mukaisesti. Kun testausvaihe on valmis, sovellus on periaatteessa valmis julkaistavaksi. (Kasurinen, 2014, Luku 1 Testaaminen yleisesti, Mitä testaustyö tarkoittaa)

Toinen tunnettu peräkkäisten vaiheiden sovelluskehitysmalli on V-malli. Siinä sovelluskehitys etenee vesiputousmaiseen tyyliin vaiheittain eteenpäin, mutta testauksen eri tasoja on liitetty jokaiseen vaiheeseen mukaan projektin alusta lähtien. Tämä tuo huomattavan edun verrattuna vesiputousmallin kehitykseen, sillä testaus ei ole ainoastaan omana vaiheenaan kehityksen lopussa, vaan se on otettu osaksi projektin toimintaa heti alkuvaiheista lähtien. (Homès, 2012, s. 45)

Inkrementaalisessa sovelluskehityksessä projekti jaetaan pienempiin osiin toiminnallisuuksien mukaan. Sovellusta kehitetään ja laajennetaan vaiheittain kohti valmista tuotetta. Kehitettävien osien koko voi myös vaihdella suuresti (ISTQB, 2018). Testauksen rooli toistuu jokaisen kehitysvaiheen kohdalla, jossa uusi osa toiminnallisuutta liitetään osaksi kehitettävää sovellusta (Homès, 2012, s. 49).

Iteratiivisessa sovelluskehityksessä projektin vaiheet on yhdistetty ja niiden toteutus tapahtuu aina ennalta määritetylle joukolle ominaisuuksia. Määrittely, suunnittelu, toteutus ja testaus toteutuvat yhdessä tietyn ajanjakson aikana. Ajanjaksoja kutsutaan usein myös kierroksiksi, sprinteiksi tai iteraatioiksi. Kierroksien aikana voidaan kehittää joko uusia ominaisuuksia tai korjata aiemmilla kierroksilla mukana olleita ominaisuuksia. Kukin kierros kasvattaa toimivaa sovellusta askeleen eteenpäin. Kehitys jatkuu, kunnes kaikki halutut toiminnallisuudet on saatu valmiiksi tai kehitystyö lopetetaan. (ISTQB, 2018)

Testaus tapahtuu iteratiivisessa kehityksessä usealla eri tasolla projektin koko elinkaaren läpi. Tavoitteena on testata jokainen ominaisuus monella eri testaustasolla. Ketterät menetelmät kuuluvat iteratiivisen sovelluskehitysmallin piiriin, joista tunnetuimpia ovat Rational Unified Process (RUP), Scrum, Kanban ja SAFe. (ISTQB, 2018)

2.3 Testaustasot

Luvussa 2.2 kuvattu testauksen toteutus projektin eri vaiheissa sisältää testaustehtäviä, jotka määräytyvät kehitettävän sovelluksen valmiusasteen mukaan (ISTQB, 2018). Sovelluksen tila määrittää myös tason, jolla testaustehtäviä suoritetaan. Nämä testaustasot koostuvat neljästä pääluokasta, jotka ovat yksikkötestaus, integrointitestaus, järjestelmätestaus sekä hyväksymistestaus (Kasurinen, 2014, Luku 3 Testaustasot).

2.3.1 Yksikkötestaus

Yksikkötestaus on yleisin testaustaso, jossa sovelluskehittäjän luoma yksittäinen sovelluksen elementti testataan heti sen valmistuttua (Kasurinen, 2014, Luku 3 Testaustasot, Yksikkötestaus). Käytännössä yksikkötestaus toimii niin, että testauksessa olevalle elementille annetaan erityyppisiä syötteitä, joiden perustella elementti tuottaa tulosteita (Juvonen, 2018, Ohjelmistoprojektien projektimallit ja toteutusmenetelmät, Ohjelmistotestaus). Yksikkötestauksen avulla pyritään löytämään mahdollinen virhe elementistä kehityksen aikaisessa vaiheessa, jotta virheet eivät kasautuisi testauksen seuraaville tasoille (ISTQB, 2018). Kehitettävän elementin ominaisuuksista riippuen testaukseen voidaan tarvita erillisiä testikomponentteja, koska elementit kehitetään erillisinä

osina sovellusta, eivätkä ne vielä pysty suoriutumaan itsenäisesti toiminnoista (Kasurinen, 2014, Luku 3 Testaustasot, Yksikkötestaus).

Yksikkötestaus suoritetaan lähtökohtaisesti elementin kehityksen yhteydessä tai heti sen valmistuttua sovelluskehittäjän toimesta. Testaaja voi olla myös muu kuin kehittäjä, mutta tässä tapauksessa testaajalla tulee olla pääsy elementin lähdetietoihin. (ISTQB, 2018)

2.3.2 Integrointitestaus

Yksikkötestausvaiheen jälkeen siirrytään seuraavalle testaustasolle, joka on integrointitestaus. Tässä vaiheessa sovelluselementtejä liitetään yhteen sekä varmistetaan, että liitokset ja niiden toiminnallisuudet toimivat määrittelyjen mukaisesti (Juvonen, 2018, Ohjelmistoprojektien projektimallit ja toteutusmenetelmät, Ohjelmistotestaus). Yhteen liitettyjen elementtien testauksessa keskitytään laajempien kokonaisuuksien toimivuuteen, koska yksittäisten elementtien toimivuus pitäisi olla varmistettu jo yksikkötestauksen vaiheessa (ISTQB, 2018). Kaikkia liitoksia ei tehdä samalla kertaa, vaan osia lisätään sekä testataan yksi kerrallaan samalla varmistaen, että sovellus säilyttää toimivuutensa (Kasurinen, 2014, Luku 3 Testaustasot, Integrointitestaus). Elementtien kehitys on monissa ohjelmistoprojekteissa jaettu monen eri kehittäjän välille ja integrointitestaus paljastaa usein mahdolliset näkemyserot suunnittelijoiden ja kehittäjien välillä (Juvonen, 2018, Ohjelmistoprojektien projektimallit ja toteutusmenetelmät, Ohjelmistotestaus).

2.3.3 Järjestelmätestaus

Kun yksittäisten elementtien ja niiden liitosten toimivuus on testattu, siirrytään koko sovelluksen toimintaa kattavan järjestelmätestauksen pariin. Järjestelmätestauksen tavoitteena on varmistaa, että kaikki aiemmin luodut sovelluksen elementit ja niiden väliset integraatiot toimivat määrittelyjen mukaisesti yhtenä kokonaisuutena (Kasurinen, 2014, Luku 3 Testaustasot, Järjestelmätestaus). Järjestelmätestaus pyritään suorittamaan tuotantoympäristöä muistuttavassa testiympäristössä (ISTQB, 2018). Järjestelmätestausvaiheen jälkeen sovellus esitellään asiakkaalle, joten sen vuoksi on erityisen tärkeää, että toiminnallisuudet ovat testattu tuotantoa muistuttavassa ympäristössä (Calvello, 2022). Järjestelmätestauksen havaintojen perusteella sovelluksen

elementtejä ja toiminnallisuuksia voidaan joutua muokkaamaan vielä paljonkin (Kasurinen, 2014, Luku 3 Testaustasot, Järjestelmätestaus).

2.3.4 Hyväksymistestaus

Viimeisessä testausvaiheessa ennen sovelluksen luovuttamista asiakkaalle suoritetaan sovelluksen virallinen hyväksymistestaus (Kasurinen, 2014, Luku 3 Testaustasot, Hyväksymistestaus). Hyväksymistestauksessa varmistetaan sovelluksen toimivuus sekä peilataan sovelluksen toiminnallisuuksia projektin alussa laadittuihin asiakasvaatimuksiin (Juvonen, 2018, Ohjelmistoprojektien projektimallit ja toteutusmenetelmät, Ohjelmistotestaus).

Lähtökohtaisesti asiakas tai asiakkaan edustaja tarkistaa sovelluksen toiminnallisuudet ja hyväksyy tuotteen, mikäli sovellukselle asetetut vaatimukset täyttyvät (Vala Group, n.d.a.). Hyväksymistestauksessa ei tarkoituksenmukaisesti etsitä virheitä sovelluksesta, vaan huomio kiinnittyy enemmän toiminnallisuuksiin ja siihen, pystyvätkö käyttäjät suorittamaan vaatimusten mukaiset työt (Homès, 2012, s. 64). Mikäli testaus on suoritettu kunnolla projektin aikana aiempien tasojen kohdalla, ei suuria muutostarpeita tai virheitä pitäisi hyväksymistestauksessa enää tulla esille (Juvonen, 2018, Ohjelmistoprojektien projektimallit ja toteutusmenetelmät, Ohjelmistotestaus).

2.4 Testiautomaatio

Testiautomaatio on toimintaa, missä testaus suoritetaan sovelluksen testaamista varten luoduilla automaatiotyökaluilla (Kasurinen, 2014, Luku 4 Testausmenetelmät ennen julkaisua, Automaatiotestaus). Manuaalitestauksessa testaaja varmistaa ohjelman toimintaa loppukäyttäjän näkökulmasta ja yrittää löytää mahdollisia virheitä (Garousi & Elberzhager, 2017). Tyypillisesti manuaalitestauksessa käydään läpi erilaisia käyttäjätarinoita, missä sovellukselle annettujen syötteiden toimintaa ja niiden tuottamia tuloksia tarkastellaan sekä suorituksen aikana että sen jälkeen (ATR Soft, 2018). Testiautomaation avulla testaaja hyödyntää testaukseen kehitettyjä työkaluja ja skriptejä, jolloin edellä kuvatut manuaalitestauksen vaiheet voidaan suorittaa automaattisesti ilman ihmisen suorittamia toimenpiteitä (Garousi & Elberzhager, 2017). Manuaalitestauksen työt hoitaa

testiautomaatiota hyödyntävä ohjelmistorobotti, joka suoriutuu testaustehtävistä merkittävästi nopeammin kuin manuaalitestausta suorittava ihminen (ATR Soft, 2018).

Testitapaukset, jotka toistuvat usein ja sisältävät samanlaiset vaiheet kerta toisensa jälkeen, ovat sopiva kohde testiautomaation käyttöönotolle. Toistuvien testitapausten automatisointi vapauttaa testaajien resursseja projektin muihin testausvaiheisiin. Esimerkiksi yksikkö- ja integrointitestauksessa kehitetyt sovelluksen elementit ja elementtien rajapinnat ovat yleinen kohde testiautomaation käytölle. Elementin muuttamisen jälkeen testaaja voi testiautomaatiota hyödyntäen tarkastaa helposti ja nopeasti muutoksen vaikutukset sovelluksen toimintaan. (Kasurinen, 2014, Luku 4 Testausmenetelmät ennen julkaisua, Testausautomaatio)

Käyttöliittymän testaus on myös toinen yleinen kohde testiautomaation hyödyntämiselle. Käyttöliittymän testaus koostuu yleensä toiminnoista kuten tiedostojen tallentamisesta tai haun suorittamisesta sivustolla. Näiden toimintojen automatisointia varten testaaja luo erillisellä ohjelmalla käsky-tai toimenpidesarjan toiminnon suorittamista varten. Suorituksen lopuksi ohjelma tekee vielä tarkastuksen vertaamalla lopputulosta etukäteen tehtyihin määrittelyihin. Testiautomaation rakentamista voidaan pitää vaatimustasoltaan ohjelmointityön kaltaisena työnä. (Kasurinen, 2014, Luku 4 Testausmenetelmät ennen julkaisua, Testausautomaatio)

2.5 Testiautomaation hyödyt ja haasteet

Testiautomaatio on hyödyllisintä ottaa osaksi kehitystyötä heti kehitysprojektin alussa, jolloin sovelluksen kehityksessä voidaan ottaa huomioon myös testiautomaation tarpeet (Vala Group, n.d.b.). Esimerkiksi verkkosivujen elementeille voidaan antaa testausta helpottavia yksilöiviä tunnisteita. Tämän avulla sivujen elementit ovat helpommin löydettävissä ja testiautomaation kehitys nopeutuu. Pienissä projekteissa testiautomaatioon panostaminen ei välttämättä ole kovin järkevää, mutta näidenkin projektien suunnitteluvaiheessa on kuitenkin hyödyllistä ottaa testiautomaation kehitysmahdollisuudet huomioon. (ATR Soft, 2018)

Testiautomaation sisällyttäminen projektiin ei automaattisesti paranna testauksen laatua. Testiautomaation hyödyt tulevat esille, mikäli siihen ollaan valmiita panostamaan. Kun testiautomaatio toimii hyvin, pystytään sen avulla vähentämään manuaalisen testauksen määrää. Toistuvat vaiheet pystytään korvaamaan tehokkaasti automatisoidulla työnkululla, mikä takaa myös tulosten paremman luotettavuuden. Työvaiheet suoritetaan aina samassa järjestyksessä ja niiden suoritustiheyttä voidaan säädellä. (ISTQB, 2018)

Testiautomaation avulla testausprosessit saadaan optimoitua niin, että testauksesta tulee mahdollisimman tehokasta, tarkkaa sekä kattavaa. Testiautomaatio on myös edellytys sovelluskehityksessä, joka perustuu jatkuvan julkaisun kehitysmalliin. (Reflector, n.d.).

Testiautomaation suosio kasvaa vähitellen. Suosiota jarruttaa kuitenkin testiautomaatioon yleisesti liitetyt väärinymmärrykset. Joidenkin mielikuvien mukaan testiautomaatio on suora korvaaja manuaalitestaukselle. Todellisuudessa testaustyö koostuu molemmista menetelmistä, jotka tilanteen mukaan täydentävät aina toisiaan. Testiautomaation tuloksia saatetaan verrata myös suoraan manuaalitestauksen tuloksiin, jolloin tulosten luotettavuus kärsii. Kustannusarviot lasketaan usein alakanttiin, minkä vuoksi testiautomaation toteuttamiselle ei todellisuudessa riitä tarpeeksi resursseja. Myös testiautomaation käyttötarkoitus arvioidaan usein väärin, jolloin sitä yritetään soveltaa asioihin, joita automaatiolla ei pystytä toteuttamaan. (Kasurinen, 2014, Luku 4 Testausmenetelmät ennen julkaisua, Testausautomaatio)

3 Testausmenetelmät

Toisessa teorialuvussa puhutaan testausmenetelmistä. Käytännön osuudessa käsiteltävät testiautomaatiotapaukset ovat luvussa esiteltävien testausmenetelmien mukaisia. Aluksi käydään läpi yleisesti web-sovelluksen testaus ja siihen liittyvät erityispiirteet, minkä jälkeen puhutaan varsinaisista testausmenetelmistä.

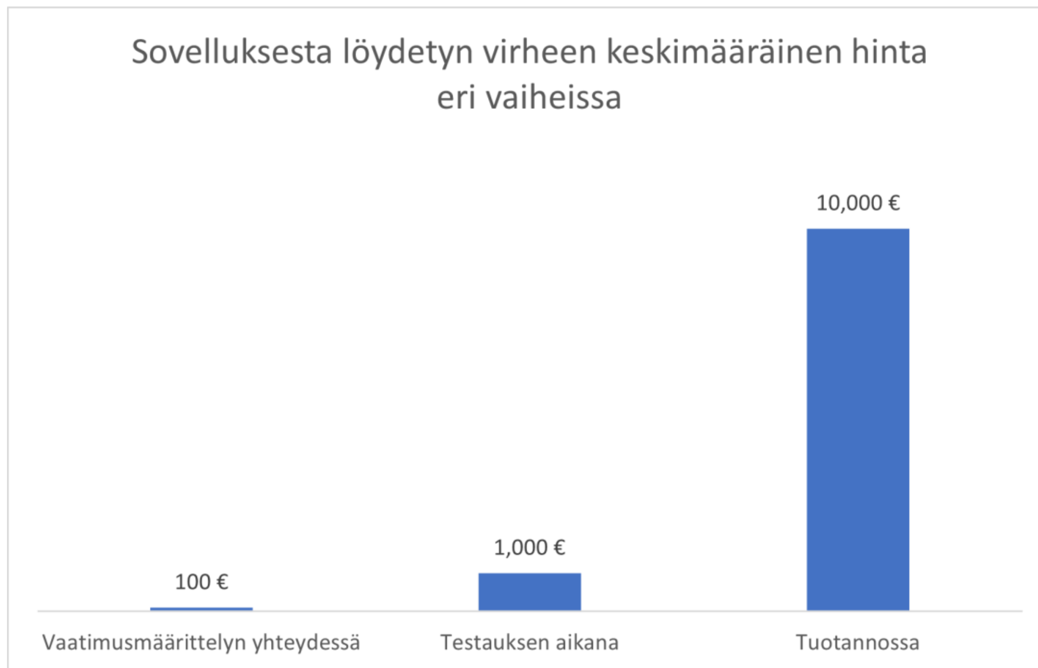
3.1 Web-testaus

Internetillä on merkittävä vaikutus elämämme eri osa-alueilla, koska yhteiskuntamme toiminnot pohjautuvat yhä enemmän web-pohjaisiin sovelluksiin (Doğan ym., 2014). Yritykset muuttavat sovelluksia yhä enemmän työpöytäversioista web-pohjaiseen ympäristöön (Cavero-Baptista, n.d.). Suurimmat hyödyt näiden web-pohjaisten sovellusten kehitysprojektien kohdalla ovat vähäiset asennuskustannukset, automaattiset päivitykset kaikille käyttäjille, sovelluksen käyttömahdollisuus jokaiselta internetiin yhteydessä olevalta koneelta sekä riippumattomuus loppukäyttäjän järjestelmästä (Doğan ym., 2014).

Web-pohjaisten sovelluksien kehitykseen liittyy myös riskejä ja haasteita. Web-sovellusten ja sivustojen pitää toimia monilla eri verkkoselaimilla, käyttöjärjestelmillä sekä laitteilla. Edellä mainitut vaihtoehdot muodostavat keskenään lukuisia eri kombinaatioita, joiden vaatimuksiin web-sovelluksen pitäisi pystyä vastaamaan. (Cavero-Baptista, n.d.)

Perusajatus web-sovelluksen testauksessa on sama kuin muussakin testauksessa. Virheet pyritään löytämään mahdollisimman varhaisessa vaiheessa sekä varmistetaan sovelluksen tai sivuston toimivuus vaatimusten ja määrittelyjen mukaisesti. Kuva 1 osoittaa myös web-sovelluksen kehityksessä pätevän faktan: mitä aiemmin virhe löydetään sovelluksesta, sitä vähemmän lisäkustannuksia yritykselle koituu. (Cavero-Baptista, n.d.).

Kuva 1 Sovelluksesta löydetyn virheen kustannusarvio sovelluskehityksen eri vaiheissa (mukaillen Cavero-Baptista, n.d.)



Hyvin toimiva web-sovellus vaatii paljon testausta. Toiminnallisuuksien manuaalinen testaaminen on hyvin työlästä ja vie runsaasti testaajan aikaa, joten web-testauksessa pyritään hyödyntämään testiautomaatiota. Testiautomaation avulla usein toistuvat tehtävät voidaan siirtää ihmisten työlistalta tietokoneen vastuulle. Testeissä vertaillaan ohjelman tuottamia tulosteita ennalta määriteltyihin, oletettuihin tulosteisiin. Näitä automatisoituja testejä voidaan ajaa jatkuvalla syötöllä tai erikseen määritetyllä aikavälillä. Testiautomaation käyttöönotto web-sovelluksen testauksessa vaatii testaajalta syvää osaamista testattavasta ohjelmistosta sekä oikeanlaiset testaustyökalut. (Cavero-Baptista, n.d.)

Web-testaus koostuu erilaisista testaustekniikoista, joita tarvitaan web-pohjaisen sovelluksen testaamiseen. Tekniikat koostuvat toiminnallisuuden testauksesta, käytettävyyden testauksesta, rajapintojen testauksesta, selaintenvälisten toimivuuden testauksesta, suorituskyvyn testauksesta, tietoturvan testauksesta sekä tietokantojen testauksesta. (Testsigma, n.d.)

Toiminnallisuuden testauksessa varmistetaan, että web-sovelluksen tai sivuston eri toiminnallisuudet toimivat sujuvasti ilman häiriöitä vaatimusmäärittelyn mukaisesti (Testsigma, n.d.). Toiminnallisuuden testauksen tapauksia voi olla esimerkiksi

sisäänkirjautumisen onnistuminen eri selaimilla ja laitteilla, lomakkeiden täyttö oikeilla tietotyypeillä sekä sivustolla olevien elementtien toiminta määrittelyjen mukaisesti (Cavero-Baptista, n.d.).

Käytettävyyden testaus on tärkeä osa web-sovelluksen testauksessa. Web-sovelluksen sujuva käyttökokemus pitää nykyiset käyttäjät tyytyväisinä sekä houkuttelee myös uusia käyttäjiä sovelluksen pariin (Testsigma, n.d.). Käytettävyyden testauksessa olennainen huomio kiinnittyy käyttäjäkokemukseen. Positiivinen käyttäjäkokemus koostuu sivuston helppokäyttöisyydestä, selkeistä ohjeistuksista, toimivasta navigaatiosta sekä tarkoituksenmukaisesta sisällöstä (Software Testing Help, 2023).

Rajapintojen testauksessa keskitytään kolmeen pääalueeseen, jotka ovat sovellus-, web- sekä tietokantapalvelimet. Tällä web-testauksen tekniikalla varmistetaan, että kyseisten palvelimien väliset yhteydet toimivat sovelluksen sisällä. Myös virhetilanteiden oikea käsittely kuuluu rajapintojen testauksen piiriin. (Testsigma, n.d.)

Selaintenvälisten toimivuuden testauksessa varmistetaan web-sovelluksen toimivuus usealla eri selaimella, käyttöjärjestelmällä sekä laitteella (Testsigma, n.d.). Esimerkiksi verkkoselaimilla ja käyttöjärjestelmillä on keskenään hyvin erilaisia asetuksia, joiden vaatimuksiin web-sovelluksen pitäisi pystyä vastaamaan (Software Testing Help, 2023).

Suorituskyvyn testauksessa varmistetaan, että web-sovellus toimii erikokoisten kuormitusten alla. Tässä testaustyypissä keskitytään erityisesti web-sovelluksen toimintaan korkean kuormituksen alaisuudessa. Testitilanteita muunnellaan kuitenkin laajemman kattavuuden saamiseksi. Suorituskykyä voidaan mitata normaalissa tilanteessa, jossa kuormaa kasvatetaan vähitellen. Suorituskykytestauksella voidaan myös selvittää murtumispiste, mihin asti kuormaa voidaan kasvattaa ennen kuin sovellus lakkaa toimimasta. (Testsigma, n.d.)

Tietoturvan testaus on tärkeä osa web-sovelluksen testauksessa etenkin arkaluontoista materiaalia käsittelevien palveluiden osalta. Tietoturvan testauksella varmistetaan, että sovelluksen ja sivustojen käyttöoikeudet toimivat oikein. Huomio tulee kiinnittyä siihen, että ulkopuoliset käyttäjät eivät pääse käsiksi sivustoihin, sovelluksen käyttöön tai arkaluontoisiin tietoihin. (Testsigma, n.d.)

Monet web-sovellukset ja sivustot ovat rakennettu tietokantojen varaan, joten niiden testaus on sen vuoksi äärimmäisen tärkeää (Testsigma, n.d.). Tietokantojen data saadaan käyttöön tietokantakyselyjen avulla ja testauksella varmistetaan näiden kyselyjen toimivuus. Tietokannoissa olevaa dataa voidaan lisätä, muokkaa tai poistaa ja testauksessa varmistetaan myös näiden toimintojen oikea vaikutus näytettäviin tietoihin (Guru, 2023 -b).

3.2 Regressiotestaus

Sovelluskehityksen aikana on tärkeä varmistaa sovelluksen toimivuus erilaisten koodi- ja toiminnallisuusmuutosten jälkeen. Olennaista on luonnollisesti selvittää, sovellukseen tehty muutos tai uusi toiminnallisuus toimii oikein. Tämän lisäksi tulee varmistaa, että sovellukseen tehty muutos ei riko mitään olemassa olevaa toiminnallisuutta. Testausta, jota suoritetaan muutosten tekemisen jälkeen sovelluksen toiminnan varmistamiseksi, kutsutaan regressiotestaukseksi. Regressiotestauksesta käytetään myös nimitystä uudelleentestaus. (Kasurinen, 2014, Luku 4 Testausmenetelmät kehitysvaiheessa, Regressiotestaus)

Regressiotestausta voidaan suorittaa myös uusien versiojulkaisujen yhteydessä, kun halutaan varmistaa kaikkien kehityksessä olleiden toiminnallisuuksien toimivuus yhdessä. Regressiotestaus ei ole ainoastaan yksittäinen testaustaso sovelluskehityksen aikana, vaan se kattaa kaiken testaustyön, jota tehdään uuden version toimivuuden varmistamiseksi. (Kasurinen, 2014, Luku 4 Testausmenetelmät kehitysvaiheessa, Regressiotestaus)

Regressiotestaus on toistuvuuden kannalta hyvin yleinen ja sopiva kohde testiautomaation käyttöönotolle. Regressiotestejä suoritetaan projektin aikana usein toiminnallisuuksien varmistamiseksi ja osaversioiden tultua valmiiksi. Sen vuoksi testejä joudutaan suorittamaan monia kertoja uudelleen, mikä onkin kenties tärkein vaatimus testiautomaation käyttöönotolle. (Kasurinen, 2014, Luku 4 Testausmenetelmät kehitysvaiheessa, Regressiotestaus)

Regressiotestaus on tärkeää etenkin projekteissa, joissa kehitys tapahtuu ketteriä menetelmiä hyödyntäen. Näihin projekteihin liittyy olennaisesti myös jatkuva integrointi, joissa toiminnallisuuksia ja koodia lisätään jatkuvasti sovellukseen sitä mukaa kun niitä valmistuu (Juvonen, 2018, Ohjelmistoprojektien projektimallit ja toteutusmenetelmät,

Ohjelmistotestaus). Jatkuvan integroinnin mahdollistaminen vaatii näihin muutoksiin pohjautuvaa automatisoitua testausta, mikä olisi hyvä sisällyttää kehitykseen jo projektin alusta lähtien (ISTQB, 2018).

Kyseiset automatisoidut testit säästävät huomattavasti aikaa verrattuna siihen, että käyttäjä joutuisi suorittamaan testivaiheet aina manuaalisesti jokaisen päivityksen jälkeen. Testit antavat myös automaattisesti valmiin raportin, josta voidaan nopeasti paikantaa mahdolliset virheet sovelluksessa. (Homann, n.d.b.)

3.3 Päästä päähän -testaus

Sovellukset ovat kehityksen edetessä muuttuneet yhä monimutkaisimmiksi järjestelmiksi. Nämä järjestelmät koostuvat monista eri komponenteista, integraatioista muiden komponenttien välillä sekä tietokannoista. Yksikin virhe yhdessä komponentissa voi aiheuttaa koko järjestelmän halvaantumisen. Päästä päähän -testauksella pyritään varmistamaan sovelluksen toimivuus yhtenä kokonaisuutena käymällä sen kaikki kerrokset läpi yksittäisestä komponentin toiminnasta aina integraatioihin ja tietokantoihin asti. Ajatuksena on tehdä samoja asioita, mitä tavallinen käyttäjä tekisi sovelluksella. Näiden perusteella luodaan käyttäjätarinoita, joiden testaaminen varmistaa sovelluksen eri kerrosten toimivuuden. (Tihekari, 2022)

Päästä päähän -testauksessa keskitytään sovelluksen toimintaan käyttäjän näkökulmasta erilaisten työkulkujen kautta. Testitapauksia voidaan suorittaa manuaalisesti seuraamalla käyttäjätarinan vaiheita, mutta suurempi hyöty saadaan käyttämällä testiautomaatiota. Etenkin suurempien projektien yhteydessä, missä sovelluksen kehitys tapahtuu jatkuvan integroinnin kautta, päästä päähän -testausta voidaan hyödyntää regressiotestauksen tapaan. Tällöin uusien versiojulkaisujen yhteydessä työnkulut voidaan testata nopeasti ja tehokkaasti testiautomaatiotyökaluja hyödyntäen ja havaita mahdolliset virheet sovelluksen eri osissa. (Schmitt, 2022)

Päästä päähän -testauksella voidaan tarjota lisäarvoa asiakkaan lisäksi myös koko projektitiimille. Testitapaukset toimivat itsessään dokumentaation lähteinä, koska ne kirjoitetaan asiakkaan näkökulmasta. Tällöin raportit ovat luettavissa ymmärrettävällä tavalla

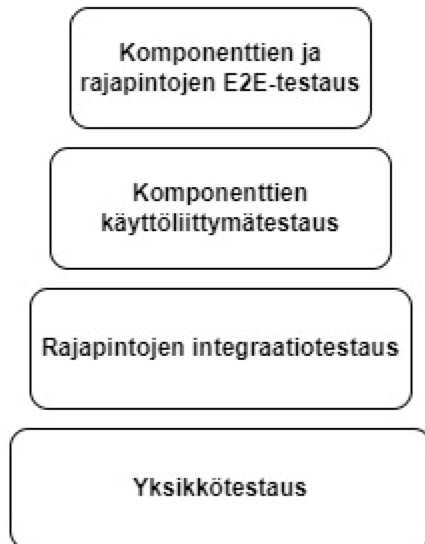
ja uusien kehittäjien on helppo ymmärtää sovelluksen toiminnan periaatteita (Tihekari, 2022). Päästä päähän -testauksella saadaan myös katettua yksikkö- ja integraatiotestejä laajempia sovelluksen osakokonaisuuksia (Schmitt, 2022).

Tähän testausmenetelmään liittyy kuitenkin myös haasteita. Testitapausten luominen voi olla hyvin hidasta ja aikaa vievää, koska testaajan tulee ymmärtää testattava sovellus läpikotaisin. Tämän lisäksi testien suunnittelu voi olla työlästä, mikäli testeissä halutaan ottaa huomioon suoritusympäristön vaatimukset. Mikäli testaaja ei myöskään ymmärrä kunnolla asiakkaan vaatimuksia sovelluksen suhteen, testaus saattaa kiinnittyä vääriin asioihin ja oikeat ongelmakohdat jäävät huomaamatta. (Schmitt, 2022)

Verkkosivun sisäänkirjautuminen voisi olla yksi esimerkki päästä päähän -testauksesta. Tässä tapauksessa käyttäjä voi syöttää lomakkeelle tyhjät kirjautumistiedot, väärät kirjautumistiedot, oikeat kirjautumistiedot sekä painaa sisäänkirjautumispainikkeesta. Päästä päähän -testauksella varmistetaan, että jokainen edellä lueteltu vaihe toimii työnkulun mukaisesti oikein. (Schmitt, 2022)

Päästä päähän -testaus toteutetaan web-testauksessa viimeisessä vaiheessa kuvan 2 mukaisesti. Ensin kehittäjät suorittavat yksikkötestauksen komponenteille, jonka jälkeen testaustasoja mukaillen varmistetaan komponenttien välisten integraatioiden toiminta. Integraatiotestauksen jälkeen varmistetaan komponenttien toimivuus sovelluksen käyttöliittymässä. Viimeisessä vaiheessa käyttöliittymäkomponenttien ja ohjelmistorajapintojen integraatioiden toiminta testataan päästä päähän. (Knight, 2022)

Kuva 2 Web-testauksen tasot testaustasoja mukaillen (mukaillen Knight, 2022)



3.4 Savutestaus

Savutestauksella varmistetaan sovelluksen perustoiminnallisuuksien toimivuus. Savutestaus on ikään kuin alustava tutkinta, joka pitää tehdä ennen tarkemman testauksen suorittamista. Perustoiminnallisuudet koostuvat hyvin yksinkertaisista asioista ja mikäli savutestaus paljastaa virheen sovelluksessa, tarkempaan testaustyöhön ei edes kannata lähteä. (Kasurinen, 2014, Luku 4 Testausmenetelmät ennen julkaisua, Savutestaus)

Savutestaus suoritetaan yleensä sovelluskehityksen aikana ennen regressiotestauksen aloittamista. Savutestaus on myös hyvä kohde testiautomaatiolle, jolloin savutestejä voidaan suorittaa säännöllisin väliajoin. Tällöin esimerkiksi tuotannossa olevan sivuston perustoiminnallisuuksien toimivuus voidaan varmistaa säännöllisesti ja mahdolliset virhetilanteet tulevat välittömästi toimintaa seuraavien tahojen tietoon. (Funk, n.d.)

4 Kehittämistyön menetelmät ja teknologiat

Opinnäytetyön tavoitteena on saada otettua käyttöön uusi web-sovelluksen testaukseen tarkoitettu kirjasto Robot Frameworkissa. Käyttöönotto käsittää tässä tapauksessa kirjaston asennuksen sekä erikseen valittujen testisarjojen korvaamisen käyttäen uutta Browser library -kirjastoa. Testisarjat ovat laadittu teoriaosassa esiteltyjen testaustasojen ja testausmenetelmien varaan.

Testisarjojen valinnoissa kiinnitettiin huomioita siihen, että niillä katettaisiin mahdollisimman laajasti sivuston eri toiminnallisuuksia. Testisarjojen valintojen lisäksi työhön päätettiin sisällyttää myös kahden eri web-sovelluksen testejä, koska kyseiset sovellukset on toteutettu eri teknologioita käyttäen. Näin olleen havaintojen pohjaksi saadaan myös uuden kirjaston toimivuus molempien sivustojen testauksen parissa.

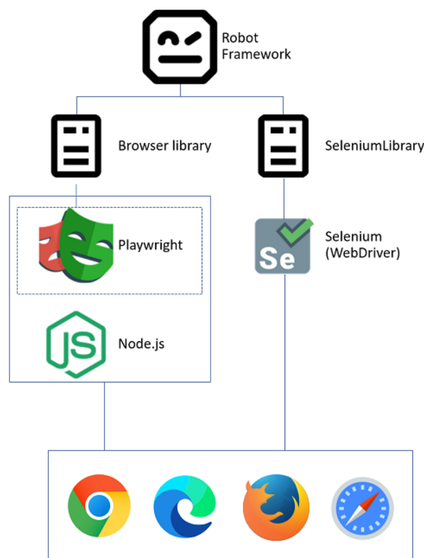
Korvaamistyön päätteeksi molempien kirjastojen testisarjoja suoritetaan muutaman kerran ja vertaillaan niiden tuloksia keskenään. Testien suoritusaika, luotettavuus (mahdollisten epäonnistuneiden testien määrä), korvaamistyöhön käytetty aika sekä korvaamistyön aikana tulleet havainnot toimivat pohdinnan materiaalina, kun mietitään mahdollista siirtymistä Browser libraryn käyttöön.

Työ toteutetaan omana kehitysprojektinaan, jossa nykyisellään käytössä olevista testisarjoista luodaan Browser librarya käyttävät uudet testisarjat. Työn etenemistä seurataan viikoittaisissa katselmuksissa sekä koululla että työpaikan puolella. Työtehtävien hallintaan ja seurantaan hyödynnetään koulun projektitöistä tutuksi tullutta Kanban-taulua. Kanban-tauluun on määritelty avainliput jokaisen luvun osalta, missä kuvataan tarkemmin kyseiseen luvun sisältö. Avainlipuille määritellään päivämäärä, johon mennessä osion tulisi olla valmis. Taulussa on lisäksi neljä eri saraketta, jotka kuvaavat avainlippujen valmiusastetta. Valitsin Kanban-taulun hallinnointiin Planner-työkalun, jonka näkymä on esitelty kuvassa 3.

Kuva 3 Kanban-taulu Microsoftin Plannerissa

Seuraavaksi luvussa esitellään teknologiat, joita työn aikana käsitellään. Kuvassa 4 nähdään teknologioiden rakenne hyvin yleisellä tasolla. Ylimmällä tasolla on Robot Framework ja sen alapuolella varsinaiset testikirjastot. Kirjastot taas pohjautuvat niiden taustalla oleviin automaatiokehyksiin, joiden avulla verkkoselaimia pystytään kontrolloimaan.

Kuva 4 Teknologiat Robot Frameworkin ja selainautomaation taustalla



4.1 Selenium

Selenium on suosittu avoimen lähdekoodin ohjelmisto, jota käytetään selainten automaatioon sekä testaukseen. Selenium tukee monia eri selaimia ja käyttöjärjestelmiä, minkä lisäksi se mahdollistaa testiautomaation toteutuksen lähes kaikilla suosituimmilla ohjelmointikielillä (Homann, n.d.a.). Seleniumin periaatteena on tarjota yhteinen rajapinta kaikille tärkeimmille selainteknologioille, joiden perusteella automaatio pystytään toteuttamaan. Seleniumin ensimmäinen versio julkaistiin jo vuonna 2004 (Selenium, n.d.a.).

Seleniumia kutsutaan sateenvarjoprojektiksi, joka koostuu eri aliprojekteista (Selenium, n.d.a.). Projektien työkalujen toiminnallisuudet eroavat hieman toisistaan, mutta niiden yhteisenä tavoitteena on varmistaa toiminallisen testauksen automatisointi mahdollisimman tehokkaalla tavalla (Peres, 2018, 3.1 Selenium). Lisäksi ne muodostavat yhdessä selainautomaation työkalupakin, minkä avulla selaininstansseja voidaan hallita etänä sekä emuloida käyttäjän toimintaa selaimella (Selenium, n.d.a.) Merkittävimmät työkalut ovat Selenium IDE, Selenium WebDriver sekä Selenium Grid (Homann, n.d.a.).

Selenium IDE on selainten lisäosa, jonka avulla voidaan nauhoittaa ja toistaa testitapauksia. Testitapaukset luodaan nauhoittamalla toiminnot, joita käyttäjä tekee selaimella. Toiminnot käännetään Seleniumin omalle kielelle, jonka jälkeen testit voidaan suorittaa työkalun avulla.

Selenium IDE on Firefoxille ja Chromelle asennettava lisäosa, jonka käyttöönotto on nopeaa ja yksinkertaista. (Homann, n.d.a.)

Selenium WebDriver on puolestaan ohjelmointirajapinta, jonka avulla testitapauksia voidaan luoda ja suorittaa (Homann, n.d.a.). Testiautomaation toteutus verkkosivuille tapahtuu tässä tapauksessa WebDriverin rajapintojen kautta. WebDriver käyttää selainvalmistajan tarjoamaa ohjelmointirajapintaa selaimen hallintaa sekä testien suorittamista varten, jolloin testien suoritus muistuttaa lähes täysin oikean käyttäjän toimintaa selaimella (Selenium, n.d.b.).

Selenium Grid on ympäristö, joka koostuu yhdestä pääjärjestelmästä sekä useista eri alijärjestelmistä. Grid:n avulla testejä voidaan suorittaa samanaikaisesti usealla eri koneella, mikä mahdollistaa myös useiden järjestelmien ja selainten muodostamien yhdistelmien testaukset samanaikaisesti (Homann, n.d.a.).

Kuva 5 havainnollistaa yhteyksien muodostamista Seleniumissa. Seleniumin yhteysprotokollana toimii HTTP-protokolla. Kun testi käynnistetään, käyttäjän kirjoittaman avainsanan Selenium-koodi käännetään JSON-muotoon. Vastauksena saatu JSON-koodi puolestaan lähetetään eteenpäin selaimen palvelimelle HTTP-protokollaa käyttäen. Jokaisen toiminnon jälkeen yhteys suljetaan ja luodaan seuraavan toiminnon kohdalla jälleen uudelleen. (Pandey, 2023)

Kuva 5 Kuvaus Seleniumin arkkitehtuurista yhteyden muodostamisen osalta (mukaillen Lambdatest, n.d.)

4.2 Playwright

Playwright on web-testauksen ja selainautomaation kehys, jolla voidaan yhden ohjelmointirajapinnan turvin automatisoida Chromium-, Firefox- ja WebKit -verkkoselainmoottoreihin perustuvia selaimia (Lambdatest, 2023). Playwright:n kehityksestä vastaa Microsoft ja se on suunniteltu erityisesti päästä päähän -testausta varten (Colantonio, 2022). Playwright perustuu Seleniumin tapaan avoimeen lähdekoodiin ja sen ensimmäinen versio julkaistiin vuonna 2020 (Lambdatest, 2023).

Playwright tukee yleisimpiä selaimia sekä käyttöjärjestelmiä (Playwright, n.d.a.). Alun perin Playwright oli JavaScript-pohjainen kirjasto, mutta kehityksen edetessä tuki on kasvanut kattamaan muitakin suosittuja ohjelmointiympäristöjä Pythonia, Javaa, .NET:iä sekä TypeScriptiä (Colantonio, 2022).

Playwright sisältää kolme sisäänrakennettua työkalua selainautomaation sekä testauksen toteuttamiseen. Ensimmäinen niistä on Codegen, jonka avulla käyttäjä luo varsinaiset testit. Käytännössä tämä tapahtuu niin, että käyttäjä nauhoittaa selaimella tehtävät toiminnot. Codegen luo toimintojen perusteella koodin, joka voidaan tallennuksen jälkeen kääntää halutulle ohjelmointikielelle. (Playwright, n.d.a.).

Playwright Inspector on graafinen työkalu, jonka avulla testejä voidaan debugata virheiden varalta. Lisäksi Inspectorilla voidaan luoda ja muokata reaaliaikaisesti paikantimia, jotka määrittävät yksittäisen elementin paikan verkkosivulla. (Playwright, n.d.b.)

Kolmas työkalu on nimeltään Trace viewer. Trace viewer on graafinen työkalu, jonka avulla käyttäjä pystyy käymään testien vaiheet läpi niiden suorituksen jälkeen. Käyttäjä pystyy näkemään visuaalisella tasolla jokaisen toiminnon vaikutuksen testin suorituksen aikana. (Playwright, n.d.c.)

Yhteyksien muodostaminen Playwright:ssa eroaa merkittävästi Seleniumista. Kuva 6 kuvaa yksinkertaisella tavalla arkkitehtuuria, jonka mukaan Playwright muodostaa testin suorituksen aikana yhden WebSocket-yhteyden. Tämä yhteys pysyy auki koko testin suorituksen ajan, minkä aikana käsitellään jokaisen toiminnon lähettämät pyynnöt. Tämä

lyhentää testien suoritusaikaa merkittävästi ja komentojen suoritus tapahtuu myös nopeammin. (Pandey, 2023)

Kuva 6 Kuvaus Playwrightin arkkitehtuurista yhteyden muodostamisen osalta (mukaillen Lambdatest, n.d.)

4.3 Node.js

Node.js on avoimen lähdekoodin alusta JavaScript-pohjaisille sovelluksille. Aiemmin JavaScript-koodia pystyttiin suorittamaan ainoastaan selaimella, mutta Node.js:n julkaisun myötä koodin suorittamiselle syntyi oma ympäristö. Tämä mahdollisti myös JavaScriptin hyödyntämisen backend-kehityksen puolella, koska aiemmin JavaScript-koodaus oli mahdollista vain frontend-kehityksen parissa. Node.js mahdollistaa näin frontend- ja backend-sovellusten kehittämisen yhdellä ohjelmointikielellä. (Semah, 2022)

Node.js:ää käytetään erityisesti skaalautuvien ohjelmistojen kehityksessä, missä ohjelmiston pitäisi hallita useita samanaikaisia tiedonsiirtopyyntöjä. Node.js suorittaa I/O-tehtävät nopeasti yhdessä ketjussa, jolloin ohjelman suoritus ei pysähdy missään vaiheessa. Tämä säästää sekä koneen resursseja että nopeuttaa tehtävien suorittamista. (Kinsta, 2023)

Playwright on Node.js:ään pohjautuva avoimen lähdekoodin kirjasto, jota tarvitaan myös Browser library -kirjaston asennuksessa. Playwright:n, Node.js:n ja Browser libraryn yhteys on havainnollistettu kuvassa 4.

4.4 Robot Framework

Robot Framework on testiautomaatioon ja ohjelmistorobotiikkaan kehitetty avoimen lähdekoodin automaatiokehys (Robot Framework, n.d.a.). Robot Frameworkin kehityksen

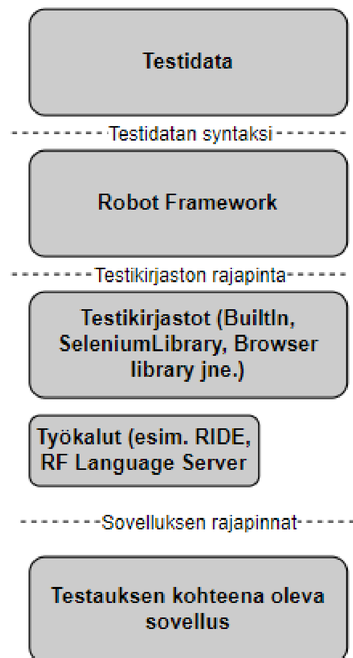
taustalla on Pekka Klärck, joka julkaisi vuonna 2005 maisterityön testiautomaatiojärjestelmistä (Copado, 2022). Robot Framework kehitettiin aluksi Nokia Siemens Networksille sisäiseksi testiautomaatiotyökaluksi, mutta jo toisen version yhteydessä Robot Framework julkaistiin avoimena lähdekoodina. Vuonna 2015 perustettiin Robot Framework Foundation, jonka tarkoituksena on tukea ja mahdollistaa Robot Frameworkin kehitystyö (Knowit, 2020).

Robot Framework on avoimen lähdekoodin periaatteiden mukaisesti avoin kaikille käyttäjille. Se on myös helposti laajennettavissa muiden sovellusten käyttöön, mikä mahdollistaa tehokkaat ratkaisut automatisointia varten (Robot Framework, n.d.a.). Robot Framework perustuu Python-ohjelmointikieleen ja sen koodi muodostuu avainsana-pohjaisista kirjastoista (Robot Framework, n.d.b.). Nämä avainsanat kertovat suoraan toiminnallisuutensa helposti ymmärrettävässä muodossa (Robot Framework, n.d.a.).

Robot Frameworkin toiminnallisuus perustuu erilaisiin testauskirjastoihin. Kirjastoja löytyy moneen eri käyttötarkoitukseen, mutta käyttäjän on mahdollista luoda myös omia kustomoituja kirjastoja. Jotkin kirjastot ovat sisäänrakennettu kehyksen ytimen kanssa. (Robot Framework, n.d.b.)

Kuva 7 havainnollistaa Robot Frameworkin arkkitehtuuria hyvin yleisellä tasolla. Ylimmällä tasolla oleva testidata on yksinkertaisessa taulukkomuodossa. Näin data on myös helposti muokattavissa. Robot Frameworkin käynnistyessä testidata otetaan käsittelyyn ja varsinaisten testitapausten suoritus aloitetaan. Kun testit on saatu suoritettua loppuun, Robot Framework luo tuloksista automaattisesti loki- ja raporttitiedostot. Kirjastot testaavat ohjelmistoa käytännössä joko rajapintojen kautta tai alemman tason työkalujen avulla. Tällöin Robot Frameworkin varsinainen ydinosa ei tiedä testattavasta ohjelmistosta mitään. (Robot Framework, n.d.b.)

Kuva 7 Robot Frameworkin arkkitehtuuri yleisellä tasolla (mukaillen Robot Framework, n.d.b.)



4.4.1 SeleniumLibrary

SeleniumLibrary on yksi Robot Frameworkin testauskirjasto, joka on kehitetty web-testausta varten. Kirjaston käyttö perustuu Selenium-automaatiokehykseen. Kirjasto käyttää Seleniumin WebDriver-moduuleja Robot Frameworkissa selaimen toimintoja varten. (SeleniumLibrary, n.d.)

SeleniumLibraryn avainsanojen käyttö perustuu verkkosivuilla olevien elementtien käsittelyyn sekä niillä tehtäviin toimintoihin. Näitä elementtejä kutsutaan paikantimiksi (kirjaston virallinen termi on *locator*). Paikantimet määrittävät elementin paikan sivulla ja ne voidaan määritellä usealla eri tavalla riippuen verkkosivun rakenteesta. Näihin määrittelyihin käytetään sivuston lähdekoodia, jossa määritetään tunnisteet elementeille. Yleisimmät tunnisteet ovat id ja name. Muita yleisiä elementtien tunnistustyylejä ovat xpath-määritykset sekä CSS-tunnisteet. (SeleniumLibrary, n.d.)

SeleniumLibraryn asennus itsessään on melko suoraviivainen ja nopea prosessi. Jotta SeleniumLibraryä pystyy käyttämään Robot Frameworkissa, täytyy käyttäjän asentaa testeissä käytettävien selaimien ajurit erikseen (SeleniumLibrary, n.d.). Esimerkiksi jos käyttäjä haluaa tehdä testit käyttäen Chrome-selainta, hänen täytyy asentaa koneelleen sen taustalla oleva Chromium-ajuri.

4.4.2 Browser library

Browser library on SeleniumLibraryn tapaan Robot Frameworkin testauskirjasto selainautomaatiota varten. Sen tarkoituksena on päivittää web-testaus 2020-luvulle (Github, 2023a). Kirjaston käyttö perustuu Playwright-automaatiokehykseen, jonka avulla selaimien hallinta toteutetaan ohjelmointirajapintaa hyödyntäen (Browser Library, n.d.). Automaation toteutus on mahdollista kolmelle selainajurille: Chromiumille (Chrome ja Edge), Firefoxille sekä WebKitille (Safari). (Github, 2023a)

Browser libraryn avainsanojen käyttö perustuu myös verkkosivun elementtien käsittelyyn. Elementtien käsittelyn mahdollistaa paikantimet (kirjaston virallinen termi on *selector)*, joiden avulla elementit löydetään sivustolta. Browser libraryn oletuksena on käyttää elementtien CSS-tunnisteita paikantimien määrittämistä varten. Browser library tukee kuitenkin myös xpath-määrityksiä, id-attribuutteja sekä text-syntaksia. (Github, 2023a)

Browser libraryn toiminta koostuu kolmesta tasosta, jotka ovat Browser, Context ja Page. Browser toimii pohjana selaimen käynnistämiselle. Browser määrittää sen, mitä selainajuria testi tulee käyttämään. Browser-prosessi käynnistyy oletuksena ns. näkymättömässä tilassa, jolloin testin suorituksen aikana selaimen graafista käyttöliittymää ei voida seurata. (Github, 2023a)

Context vastaa yhden browser-profiilin hallinnasta. Contextin sisällä voidaan jakaa evästeitä, profiiliasetuksia ja välimuistia. Contextin avulla voidaan testata kätevästi eri käyttäjäskenaarioita samalla verkkosivulla yhden contextin sisällä, jolloin selainmoottoria ei tarvitse käynnistää uudelleen skenaarioiden välillä. Contextille voidaan määritellä esimerkiksi maantieteellinen sijainti, kieli, selainikkunan koko tai väripaletti. Yksi context voi sisältää useita eri Pageja. (Github, 2023a)

Page koostuu varsinaisesta sivuston sisällöstä. Page voidaan määritellä myös selaimen välilehdeksi. Jos testissä avataan uusi Page ilman Browseria ja Contextia, kyseiset instanssit luodaan automaattisesti oletusarvoilla ennen uuden Pagen avaamista. (Github, 2023a)

Browser libraryn asennus on myös hyvin yksinkertainen prosessi. Suurimpana erona SeleniumLibraryn asennukseen verrattuna on selainajureiden automaattinen asennus, jolloin pelkällä kirjaston asennuksella selaimet ovat heti käytössä testejä varten. Kirjaston asennus onnistuu myös ilman selainajureiden asennuksia, mutta tällöin käyttäjän on itse pidettävä huolta ajureiden ylläpidosta. Browser library vaatii toimiakseen Pythonin ja Node.js:n asennukset. (Browser library, n.d.)

4.5 Robot Frameworkin testidatan rakenne ja kirjoitussyntaksi

Robot Frameworkin testitapaukset luodaan yleensä robot-päätteisiin tiedostoihin. Yksi robot-tiedosto muodostaa niin kutsun testisarjan (*Test Suite*), joka koostuu monesta eri testitapauksesta (*Test Cases*). Testitiedostot voidaan jaotella vielä omiin kansioihin, jolloin muodostuu korkeamman tason testisarjoja. Testitapausten määrälle ei ole määritelty maksimiarvoa yhden testisarjan sisällä, mutta suosituksena mainitaan korkeintaan kymmenen testitapausta yhdessä sarjassa. (Robot Framework, n.d.b.)

Testidata jaotellaan testitiedostossa omiin alueisiin taulukon 1 mukaisesti. Robot Framework tunnistaa datan tyypin alueen otsikon perusteella. Suositeltu tapa otsikon määrittelyyn on käyttää tähtimerkkejä sanan ympärillä. **Tasks** ja **Test Cases** toteutetaan samalla tyylillä testidatan sisällä ja yksi testitiedosto voi sisältää ainoastaan toisen alueen näistä kahdesta. (Robot Framework, n.d.b.)

Taulukko 1 Testitiedoston rakenne otsikon ja käyttötarkoituksen mukaan (Robot Framework, n.d.b.)

Alueen otsikko	Käyttötarkoitus
****** Settings ******	Testikirjastojen, resurssi- ja muuttujatiedostojen tuominen testitiedoston käyttöön sekä

	testitapausten/sarjojen metadatan määritys.
*** **Variables** ***	Määritellään testidatan muuttujat, jota muut alueet voivat datan sisällä käyttää.
*** **Test Cases** ***	Testitapausten kirjoitus kirjastojen avainsanoja käyttäen.
*** **Tasks** ***	Samanlainen käyttötarkoitus kuin Test Cases -osiossa. Erona on, että Tasks on tarkoitettu RPA:n (ohjelmistorobotiikan) toteutukseen.
*** **Keywords** ***	Kirjastojen avainsanoihin perustuvien omien avainsanojen alue.
*** **Comments** ***	Alue, johon voi liittää omia kommentteja tai dataa. Robot Framework jättää tämän alueen huomioimatta.

Testien kehittämisessä on hyvä noudattaa käytäntöjä, joiden avulla selkeytetään testien lukemista, ylläpitoa sekä kehittämistä. Testisarjat ja testitapaukset sarjojen sisällä olisi hyvä nimetä toimintoja kuvaavalla tavalla. Sarjojen ja tapausten osioissa sisällä on suotavaa myös avata toimintoja dokumentaatio-osiossa sekä lisätä testien sisälle kommentteja, mikäli toiminnot vaativat tarkempaa selitystä. Kustomoidut avainsanat tulisi nimetä lyhyesti ja ytimekkäästi niin, että avainsanan nimi kertoo suoraan, mitä kyseinen avainsana tekee. (Github*,* 2023b)

Taulukko 1 kuvaa testidatan rakennetta yhden tiedoston sisällä. Projekteissa on kuitenkin hyvä ottaa huomioon uudelleenkäytettävyys etenkin avainsanojen ja muuttujien kohdalla. Muuttujat lisätään usein omaan tiedostoon, jolloin testitapauksia kirjoittaessa ne saadaan käyttöön määrittämällä tiedoston polku ja nimi **Settings**-osiossa. Omaan tiedostoon kerätään usein sellaiset muuttujat, joita käytetään useammassa eri testisarjassa. Myös omat kustomoidut avainsanat kerätään usein omaan resurssitiedostoon, jotka saadaan testisarjoissa käyttöön **Settings**-osiossa. Nämä avainsanat koostuvat korkeamman tason toiminnoista, joita käytetään niin ikään useassa eri testisarjassa. (Robot Framework, n.d.b.)

Space Separated Format on yleisin tapa käsitellä testidataa Robot Frameworkilla. Tämä tarkoittaa sitä, että datan eri osiot erotetaan toisistaan vähintään kahdella eri välilyönnillä.

Esimerkiksi avainsanan käytössä testiin kirjoitetaan ensiksi itse avainsana, jonka jälkeen lisätään välilyönnit ja välilyöntien perään avainsanan vaatimat argumentit. (Robot Framework, n.d.b.)

Testidatan käsittelyyn on olemassa myös pari muuta tyyliä, jotka esitellään Robot Frameworkin User Guide -sivustolla. Työssä käsiteltävissä testitapauksissa käytetään kuitenkin ainoastaan Space Separated Format -tyyliä, joten muiden tyylien esittely on rajattu tässä kohtaa pois. Ohjelmakoodissa 1 nähdään Space Separated Format datan käsittelyssä.

Ohjelmakoodi 1 Testidatan käsittelyn syntaksi Space Separated Format -tyyliä käyttäen

```
# Space Separated Format
*** Settings ***
Documentation   Esimerkki datan käsittelystä kolmella välilyönnillä.
Library         Browser

*** Test Cases ***
Esimerkki Testi
    # Avainsanan ja argumentin välissä on 3 välilyöntiä
   New Page   https://www.hamk.fi
```

5 Ohjelmistojen asennus ja käyttöönotto

Käytännön osuuden ensimmäisessä luvussa kuvataan Robot Frameworkin sekä Browser libraryn asennusvaiheet läpi Windows-koneella. Projektityössä käytetään Robot Frameworkia kahdessa eri kehitysympäristössä, jotka ovat RIDE sekä Visual Studio Code. RIDEn asennus kuvataan luvussa läpi, mutta Visual Studio Coden osalta kuvataan ainoastaan Robot Frameworkin lisäosan asennus. Tekstiosuudessa kuvataan yleisellä tasolla asennusvaiheet ja sen yhteydessä huomioitavat asiat. Työn liitteistä löytyy tarkemmat asennusohjeet kuvakaappausten kanssa.

5.1 Pythonin ja Robot Frameworkin asennus

Robot Framework perustuu Python-ohjelmointikieleen, joten Python pitää olla asennettuna koneelle ennen Robot Frameworkin asennusta. Windows-koneella Python on suositeltavaa myös lisätä koneen ympäristömuuttujiin, jolloin Pythonin käyttöä voidaan hyödyntää komentoriviltä käsin (Robot Framework, n.d.b.). Sen voi nykyään tehdä suoraan Pythonin asennusvaiheessa, mutta jälkikäteen sen voi käydä lisäämässä Järjestelmämuuttujat-valikosta (Environment variables). Pythonin asennus on onnistunut, mikäli komentokehotteella suoritettu komento 1 antaa vastauksena asennetun Python-ohjelman versionumeron.

Komento 1 Pythonin versionumeron sekä asennuksen onnistumisen tarkastuskomento

```
python --version
```

Robot Frameworkin asennus onnistuu hyvin suoraviivaisesti komentokehotteen kautta Pythonin pip-paketinhallintajärjestelmää hyödyntäen. Yhdellä komennolla pip asentaa viimeisimmän version Robot Frameworkista. Komentokehotteen kautta onnistuu myös Robot Frameworkin päivittäminen uudempaan versioon sekä tietyn versionumeron asennus. Tämän osion toteutushetkellä viimeisin versio Robot Frameworkista oli 6.0.2 ja Pythonista 3.10.11. Robot Frameworkin uusin versio vaatii toimiakseen vähintään Pythonin 3.6-version asennuksen.

5.2 Node.js

Node.js:n asennus on yksi Browser libraryn vaatimuksista ennen sen käyttöönottoa ja asennusta. Playwright:n dokumentaatio vaatii työn toteutushetkellä asennusversioksi vähintään versionumero 16:sta. Työssä valittiin asennusversioksi 18.15.0, mikä oli sivuston suositus useimmille käyttäjille (LTS-versio). Asennus sujui mutkattomasti ohjelmiston graafisen käyttöliittymän kautta.

Node.js on lisättävä Pythonin tapaan järjestelmämuuttujiin, jotta Browser libraryn käyttö onnistuu. Tämän voi määritellä asennuksen aikana tai Pythonin tapaan muokkaamalla järjestelmämuuttujia käsin.

5.3 Browser libraryn asennus

Kun esivaatimukset on täytetty, voidaan varsinainen testikirjasto asentaa koneelle. Browser libraryn voi asentaa koneelle kahdella eri tyylillä: joko selaimien binääritiedostojen kanssa tai ilman niitä. Suositeltu tapa on asentaa binääritiedostojen kanssa, jolloin selainajureiden päivityksistä ei myöskään tarvitse itse huolehtia. Ajureiden päivitys tapahtuu kirjaston päivityksen yhteydessä. Browser libraryn asennus tehdään tässä työssä edellä mainitulla tyylillä. Asennushetkellä kirjaston uusin versionumero oli 16.0.2.

Asennus suoritetaan kahdessa vaiheessa komentokehotteen kautta. Ensimmäiseksi suoritetaan kirjaston varsinainen asennus. Tämän jälkeen kirjasto alustetaan käyttökuntoon. Kun asennus ja alustus on suoritettu, kirjasto on valmis käyttöönotettavaksi. Kirjaston asennusvaiheessa kannattaa sulkea ohjelmat, joita testien kehityksessä käytetään. Työpaikan ympäristöön asentaessa törmäsin ongelmaan, jossa Browser library ei toiminut asennusvaiheiden jälkeen. Ongelma ratkesi testitapausten kehityksessä käytetyn Visual Studio Coden uudelleenkäynnistämisellä, jonka jälkeen kirjasto saatiin onnistuneesti käyttöön.

Kirjaston päivitys tehdään kolmessa vaiheessa komentokehotteella. Ensimmäiseksi suoritetaan kirjaston varsinainen päivityskomento. Tämän jälkeen poistetaan kirjaston vanhat node-riippuvaisuudet sekä selainten binääritiedostot, jotka ovat olleet sidoksissa

käytössä olleeseen versioon. Viimeisenä vaiheena kirjasto alustetaan ja uudet riippuvaisuudet binääritiedostoineen otetaan käyttöön.

5.4 Kehitysympäristöjen asennus

Viimeisenä vaiheena asennetaan kaksi Robot Frameworkin kehitysympäristöä. Ensimmäinen niistä on nimeltään RIDE. Mikäli käyttäjä haluaa jatkaa komentorivi-pohjaisten asennusten linjalla, RIDE sopii tähän tarkoitukseen mainiosti. RIDEn asennus tapahtuu pip-järjestelmää hyödyntäen yhdellä komentokehotteen komennolla. Kyseinen komento asentaa aina sovelluksen uusimman version, joka on työn toteutushetkellä 2.0.5. Samalla komennolla onnistuu myös RIDEn päivitys uudempaan versioon. Asennuksen jälkeen RIDE voidaan avata komentokehotteen kautta, mutta RIDEn asetuksista käyttäjä voi lisätä myös pikakuvakkeen ohjelman avaamista varten.

Työpaikalla käyttämäni kehitysympäristö on Visual Studio Code. Testien kehitystyötä varten hyödyllinen lisäosa on Robocorpin Robot Framework Language Server, joka mahdollistaa testien ajamisen suoraan kehitysympäristöstä käsin. Lisäksi testien kirjoittaminen on helpompaa, kun lisäosa tarjoaa apua testikoodin kirjoitussyntaksin kanssa. Visual Studio Code oli asennettu ennakkoon sekä työpaikan koneelle että omalle kotikoneelle, joten tässä osiossa kerrotaan ainoastaan Language Server -lisäosan asennuksesta.

Lisäosan asennus on hyvin helppoa ja suoraviivaista. Lisäosa löytyi Extensions-ikkunasta hakusanalla Robot Framework, jonka jälkeen yhdellä klikkauksella asennus saatiin suoritettua valmiiksi. Lisäosalla on myös omat vaatimuksensa Pythonin ja Robot Frameworkin versioille, jotka löytyvät lisäosan omalta sivulta Visual Studio Codessa.

6 Testien korvaaminen

Käytännön osuuden toisessa luvussa käydään läpi prosessia, jossa korvataan erikseen valittuja SeleniumLibrary-kirjastoa hyödyntäviä testitapauksia Browser library -kirjaston avainsanoilla. Kappaleissa käydään ensin läpi testattava web-sovellus ja testien rakenne, jonka jälkeen keskitytään varsinaiseen testien toteutukseen. Toteutuksen jälkeen analysoidaan vielä testien eroja suoritusaikojen ja luotettavuuden suhteen.

Testien korvaamisessa käytetään kahta eri tyyliä. Ensimmäisessä tyylissä SeleniumLibraryn avainsanat korvataan suoraan testeihin ja sitä hyödyntäviin resurssitiedostoihin Browser libraryn vastaavilla avainsanoilla. Toisessa tyylissä luodaan oma resurssitiedosto, johon kirjoitetaan jokaista testeissä käytössä olevaa SeleniumLibraryn avainsanaa vastaava kustomoitu avainsana. Näihin avainsanoihin luodaan sama toiminnallisuus käyttäen Browser libraryn avainsanoja. Testitiedoston asetuksissa muutetaan viitaukset SeleniumLibrarystä Browser libraryyn, jolloin testi käyttää resurssitiedostoon luotuja kustomoituja avainsanoja.

6.1 Avainsanojen korvaaminen suoraan testeihin

Ensimmäisen korvaustyylin testauskohteena toimii web-sovellus Tekla Warehouse (https://warehouse.tekla.com). Sivusto toimii eräänlaisena varastona Tekla Structures -rakennussuunnittelun tietomallinnusohjelmistolle. Warehousesta käyttäjä voi ladata sisältöä, jota hyödynnetään Tekla Structures -ohjelmiston käytössä. Warehouse sisältää integraation varinaiseen työpöytäsovellukseen, mutta työssä käytettävissä testeissä keskitytään puhtaasti web-sivuston käyttöliittymän toiminnan testaukseen.

Testien rakenne koostuu testitiedostoista, avainsanoja sisältävästä resource-tiedostosta sekä muuttujia sisältävästä variables-tiedostosta. Lisäksi Warehousen testit käyttävät myös yhteistä resource-tiedostoa. Kyseinen tiedosto sisältää avainsanoja, joita hyödynnetään monen eri palvelun testitapauksissa.

Käytännön työ lähti liikkeelle ottamalla kopiot testikansiosta sekä siihen liittyvistä resource-tiedostoista. Nämä tiedostot sisältävät testeissä käytettävät SeleniumLibrary-kirjaston avainsanat, jotka korvataan Browser library -kirjaston

avainsanoilla. Kopioidulle testikansiolle luotiin oma kansio, jonka avulla pyrittiin välttämään mahdolliset sekaannukset nykyisten testitapausten kanssa. Resource-tiedostoihin liitettiin nimen perään browser-tunnisteet. Variables-tiedosto sisältää staattisia tekstimuuttujia sekä xpath-polkuja verkkosivun elementeille, joihin ei lähtökohtaisesti tarvita muutoksia. Mikäli muutoksia joihinkin muuttujiin tulisi tehdä, luodaan kyseiselle elementille siinä tapauksessa oma muuttuja testitapauksen sisälle. Kuvassa 8 nähdään testidatan rakenne kansiopolussa.

Kuva 8 Testidatan rakenne kansiopolussa

```
robottests/
├─ TWH_Browser/
|   ├─ .robot-files
|   ├─ Downloads/
├─ resources/
|   ├─ common_browser.resource
|   ├─ warehouse_browser.resource
├─ variables/
|   ├─ twhvariables.resource
```

Testien korvaaminen tehtiin seitsemälle eri testisarjalle, jotka käsittävät yhteensä 39 testitapausta. Kyseiset testit ovat kerran vuorokaudessa automaattisesti suoritettavia järjestelmätestejä, joilla seurataan sivuston toimivuutta. Testit toimivat myös regressiotesteinä, koska ne voidaan ajaa nopeasti uusien toiminnallisuusjulkaisujen jälkeen toiminnan varmistamiseksi. Yksi testisarjoista kuuluu myös sivuston perustoiminnallisuuksien vahvistavaan savutestaussarjaan.

Korvattavat testit kattavat laajasti sivuston toiminnallisuuksia. Basic View -sarjassa varmistetaan, että sivuston perusnäkymä näyttää oikealta. Analytics Views -sarjassa varmistetaan sivuston analytiikan toimivuus käyttäjäoikeuksien mukaisesti. Content Translations-, Organization Page- sekä Edit Collection and Package -sarjoissa testataan, että sivustolla olevaa sisältöä voidaan muokata käyttäjäoikeuksien mukaisesti. Copy Content -sarjan testit varmistavat sisällön kopioimisen ja liittämisen toiseen sisältökokoelmaan. Seitsemäs testisarja (Flow 3) keskittyy sisällön luomisen, näkyvyyden ja lataamisen toimivuuden testaukseen.

Testien korvaamisen aloitin etenemällä testitapausten koodia läpi rivi kerrallaan. Aina kun vastaan tuli SeleniumLibrary-kirjaston avainsana, niin etsin vastaavan avainsanan Browser libraryn avainsanojen dokumentaatiosta ja tein tarvittavan muutoksen. Testitapaukset ovat rakennettu abstraktiotasoja hyödyntäen, mikä tarkoitti sitä, että korvattavat avainsanat löytyivät usein joko testitiedoston Keywords-osiosta tai resource-tiedostoista. Tämä aiheutti varsinkin alussa paljon siirtymistä tiedostojen välillä. Monissa kustomoiduissa avainsanoissa oli myös käytetty toista kustomoitua avainsanaa, mikä aiheutti jälleen uuden siirtymisen avainsanojen välillä.

Ensimmäinen ongelmatilanne koski kustomoitua avainsanaa, jota käytetään jokaisessa palvelussa selainympäristön käyttöönottoa varten. Seleniumiin pohjautuvassa avainsanassa käytetään **Create Webdriver** -avainsanaa, jolle annetaan kustomoidussa avainsanassa lukuisia eri argumentteja selaimen sekä käyttöympäristön mukaan. Koska tässä työssä testit ajetaan ainoastaan omalla paikallisella työympäristöllä, päätettiin korvattavissa testeissä luoda mahdollisimman yksinkertainen alustus selaimen käynnistämistä varten. Mahdollisen jatkokehityksen varaan jäi yleisen avainsanan toteutus selainympäristön käyttöönottoa varten kaikissa testiympäristöissä.

Selainympäristö koostuu Browser libraryssä kolmesta eri tasosta, jotka ovat Browser, Context sekä Page. Browser libraryn testeissä jokainen taso voidaan käynnistää avainsanalla, joka muodostuu lisäämällä termin eteen **New (New Browser, New Context, New Page)**. Mikäli käyttäjän ei tarvitse muuttaa Browserin oletusasetuksia argumenteissa, ei New Browser -avainsanaa tarvitse testissä välttämättä käyttää. Sama koskee myös Contextia: mikäli Browserin ja Contextin oletusmäärityksiin ei tarvitse muutoksia, voi käyttäjä käynnistää selaimen suoraan **New Page** -avainsanalla.

Browser library käynnistää oletuksena selaimen headless-tilassa käyttäen Chromium-verkkoselainajuria, johon Chrome sekä Edge pohjautuvat. Headless-tila tarkoittaa sitä, että testien suorituksen aikana käyttäjä ei näe selaimen graafista käyttöliittymää ollenkaan. Koska nykyiset testit suoritetaan graafisen käyttöliittymän ollessa näkyvillä ja korvaamisessa halusin myös nähdä testien suoritusvaiheet, täytyi **New Browser** -avainsanan argumentiksi lisätä **headless=False**.

Toinen lisättävä argumentti koski selaimen määritystä. Nykyisiä testejä suoritetaan Chromella, Firefoxilla ja Edgellä. Suurin osa testeistä ajetaan käyttäen Chromea, joten oletusargumentiksi lisäsin dokumentaation perusteella **browser=chromium**.

Browser library ei sisällä avainsanaa, jolla selainikkunan saa suurennettua koko ruudun kokoiseksi. Chromium-selaimet pystytään kuitenkin käynnistämään koko ruudun kokoisina. Tätä varten **New Browser** -avainsanalle pitää antaa argumenteissa **args=["--start-maximized"]**. Firefoxille löysin hieman vastaavanlaisen komennon **--kiosk**, mutta tämä luo rajoitteita selaimen toimintoja varten, joten sitä ei pystytä tässä tapauksessa käyttämään.

Mikäli testin suorituksen aikana suoritetaan tiedoston lataus ja ladattu tiedosto halutaan säilyttää testin suorituksen päätyttyä, argumentteihin tarvitaan latauskansion polun määritys. Lisäsin avainsanan argumentteihin latauskansion määrityksen oletuksena tyhjänä **${downloadDir}=${EMPTY}**. Latauskansio voidaan määritellä latausta tarvitsevissa testitapauksissa erikseen.

Contextin luomisessa vaadittiin yksi argumentti, joka liittyi selaimen käynnistämiseen koko ruudun suuruisena. Tämä argumentti on **viewport=${None}.** Tämän jälkeen määritin vielä **New Page** -avainsanan halutulla verkkosivun osoitteella **${URL}**, sekä korvasin evästeiden syöttämiseen tarvittavat muutokset kustomoidussa avainsanassa. Selaimen käynnistämisestä muodostui ohjelmakoodin 2 näköinen kustomoitu avainsana.

Ohjelmakoodi 2 Selaimen käynnistämisen kustomoitu avainsana

```
Browser Startup
    [Arguments]    ${url}    ${browser}    ${downloadDir}=${EMPTY}
    [Documentation]    Starts up browser.
    New Browser    headless=False    browser=${browser}    downloadsPath=${downloadDir}    args=["--start-maximized"]
    New Context    viewport=${None}
    New Page    ${url}
    common_browser.Inject Testuser Cookies And Reload Page
    Browser.Go To    ${url}
```

Sisäänkirjautumiseen liittyvässä kustomoidussa avainsanassa törmäsin seuraavaan ongelmatilanteeseen. Ikkunassa, jossa sisäänkirjautuminen suoritetaan, käytettiin SeleniumLibrary-testeissä kustomoitua avainsanaa **Click Element by JavaScript**. Avainsana hakee sivustolta ensin määritetyn elementin, jonka jälkeen kyseiselle elementille annetaan JavaScript-komento klikkausta varten. Browser libraryssä on vastaava avainsana **Evaluate**

JavaScript, jota yritin suoraan hyödyntää korvauksessa. Tämä ei kuitenkaan onnistunut, joten käytin uudessa testissä ainoastaan avainsanaa **Type Secret**, joka syöttää tekstin kenttiin eikä näytä syötteiden arvoja testin raportissa. Testin suoritus muuttui hieman, koska SeleniumLibrary-testeissä käyttöliittymässä ei näy tekstikenttiin kirjoitetut tekstit, kun taas Browser library -testeissä kirjoitukset ilmestyivät tekstikenttiin. Ohjelmakoodissa 3 nähdään muutokset avainsanojen välillä.

Ohjelmakoodi 3 Sisäänkirjautumistietojen syöttäminen SeleniumLibrary-avainsanoja käyttäen vs. Browser library -avainsanoja käyttäen

SeleniumLibrary-versio

```
Fill Login Info For Automation Users
    [Arguments]    ${username}    ${password}
    [Documentation]    Input user name and password in automation user login page
    [Tags]    ATC
    Run Keyword And Continue On Failure    Wait Until Page Contains    Sign in with your username and password
    Wait For Condition    ${document_ready_state}
    Wait Until Element Is Enabled    ${signInFormUsername}
    Click Element By Javascript    ${signInFormUsername}
    Execute Javascript    ${username}
    Execute Javascript    ${password}
    Wait Until Element Is Enabled    ${sign_in_button}
    Click Element By Javascript    ${sign_in_button}
    Wait Until Element Is Not Visible    ${sign_in_button}    30s
```

Browser library -versio

```
Fill Login Info For Automation Users
    [Arguments]    ${username}    ${password}
    [Documentation]    Input user name and password in automation user login page
    [Tags]    ATC
    Type Secret    ${username_field}    $username
    Type Secret    ${password_field}    $password
    Click    ${submit_button}
```

Seuraava ongelmatilanne syntyi testissä, jossa sivustolta ladataan analytiikkaa koskeva csv-tiedosto. SeleniumLibrary-testissä lataus tapahtuu automaattisesti testikäyttäjän suorittaman toiminnon jälkeen, jossa klikataan latauksen aloittavaa latauspainiketta. Tämän jälkeen varmistetaan OperatingSystem-kirjastoa hyödyntäen, että ladattu tiedosto on ilmestynyt latauskansioon.

Browser library toimii latausten kohdalla eri tavalla. Latauksia voidaan suorittaa käyttäen kahta eri avainsanaa: **Download** ja **Promise To Wait For Download**. **Download**-avainsanaa suositellaan käytettäväksi, mikäli käytössä on suora verkkosivun osoite, josta lataus suoritetaan. Mikäli lataus tapahtuu jonkun toiminnon mukaan (kuten latauspainikkeen klikkaus eikä toimintoa varten ole saatavilla osoitetta), käytetään **Promise To Wait For**

Downloadia. Mikäli Browserin luomisen yhteydessä latauskansiota ei määritetä, tiedosto tuhotaan testi päättymisen jälkeen automaattisesti. Koska testissämme ei ole käytössä suoraa osoitetta lataukselle, uudessa testissä tuli käyttää pidempää versiota.

Käytännössä latauksen suorittaminen **Promise To Wait For Download** -avainsanaa käyttäen menee niin, että kyseistä avainsanaa käytetään ennen latauksen aktivoivaa toimintoa. Avainsanan palauttama ”lupaus” tallennetaan muuttujaan, joka odottaa seuraavan lataustapahtuman suoritusta sivulla. Avainsanaan määritetään myös ladattavan tiedoston polku sekä tiedostonnimi. Tämän jälkeen latauksen aktivoiva toiminto suoritetaan ja odotetaan latauksen suoritus loppuun, jonka jälkeen testissä voidaan varmistaa latauksen onnistuminen.

Ongelmatilanne koski väärin määriteltyä tiedostopolkua. Windows-järjestelmää käytettäessä kansiopolun erotin on eri kuin UNIX-järjestelmissä. Väärä formaatti aiheutti epäonnistuneen latauksen ja testin epäonnistumisen. Robot Framework sisältää sisäänrakennetun ympäristömuuttujan **${/}**, jonka voi lisätä polun määritykseen. Tällöin erotin määritellään aina oikein käytössä olevan järjestelmän mukaisesti ja tämän määrityksen jälkeen sain latauksen myös toimimaan Browser libraryn testissä.

Merkittävä ero testien välillä liittyi elementtien tunnistamiseen sivustolla. SeleniumLibrary-testeissä käytetään avainsanoja, jotka itsessään kuvaavat tilannetta, jolle haetaan vahvistusta (esim. **Page Should Contain Text**). Browser libraryn toiminta kyseisissä tapauksissa perustuu **Get Element States** -avainsanan käyttöön. Avainsanalle annetaan argumentiksi elementin paikannin, jonka jälkeen hyödynnetään ”vakuuksia” (**Assertions**). Vakuuksien avulla tarkistetaan, sisältääkö elementti määriteltyjä ominaisuuksia. Vakuuksien tarkastamisen voi määrittää joko niin, että elementin on pakko sisältää ainoastaan argumenteissa määritelty ominaisuus tai elementti voi sisältää myös muita ominaisuuksia. Erot kirjastojen välisissä toteutuksissa nähdään ohjelmakoodissa 4.

Ohjelmakoodi 4 Erot kirjastojen toteutuksissa liittyen elementtien tunnistamiseen sivustolla

```
#SeleniumLibrary
${contains}    Run Keyword And Return Status    Page Should Contain    ${packname}
Wait Until Element Is Visible    //*[text()='${packname}']

#Browser library
${contains}    Get Element States    "${packname}"    contains    attached
Wait For Elements State    //*[text()='${packname}']    visible
```

Nykyisissä testeissä on käytössä paljon tekstiin perustuvia tunnistamisia, mikä vaati muutoksen avainsanan käytön lisäksi itse tunnistettavaan elementtiin. Näissä kohdissa täytyi alkuperäinen teksti muuttaa elementin paikantimeksi. Käytin muutoksissa xpath-määritystä sekä Browser libraryn text-syntaksia.

Browser libraryn suuri vahvuus nousi esille kohdissa, joissa nykyiset SeleniumLibrary-testit odottavat elementtien ilmaantumista sivulle ennen varsinaista toiminnan osuutta. Browser libraryä käyttäen pystyin monissa tapauksissa tekemään yhdellä avainsanalla saman toiminnon, johon SeleniumLibrary-testien kohdalla tarvittiin monta riviä koodia. Browser library odottaa automaattisesti elementin ilmaantumisen sivulle ja vahvistaa, että se on klikattavassa ennen toiminnon suorittamista. SeleniumLibraryä käytettäessä täytyy omilla avainsanoilla vahvistaa, että elementti on sekä ilmaantunut sivustolle että on käytettävissä toimintoja varten.

Testeissä oli käytetty paljon kustomoitua avainsanaa **Click Element With Wait**. Kyseisessä avainsanassa navigoidaan ensiksi argumentissa määritetyn elementin kohtaan sivustolla. Sen jälkeen vahvistetaan kahdella avainsanalla, että elementti on näkyvissä ja aktivoituna. Tämän jälkeen keskitetään vielä erikseen järjestelmän huomio elementtiin, jonka jälkeen varsinainen klikkaus suoritetaan. Korvaavissa testitapauksissa riitti ainoastaan **Click**-avainsanan käyttö, joka automaattisesti vahvisti, että elementti on näkyvillä ja klikattavissa. Näiden jälkeen testi pystyi suorittamaan toiminnon ilman erillisiä vahvistuksia.

Alussa korvaustyöhön kului paljon aikaa, koska Browser libraryn avainsanat olivat uusia ja niiden toiminnallisuuksia täytyi lukea tarkkaan dokumentaatiosta. Lisäksi lukuisat siirtymiset korvaustyön aikana tiedostojen välillä aiheutti välillä sekaannusta. Ensimmäisen testisarjan

korvaamiseen kului aikaa noin kolme työpäivää. Työ kuitenkin nopeutui erittäin paljon, kun toiminnallisuudet alkoivat olemaan tuttuja ja uudelleen eteen tulleita kustomoituja avainsanoja oli jo korvattu. Viimeisen testisarjan korvaamiseen menikin lopulta alle kolme tuntia. Kaikki SeleniumLibraryn avainsanat, jotka työssä korvattiin Browser libraryn avainsanoilla, ovat koostettu taulukkoon työn liitteeksi.

6.2 Testien korvaaminen käyttäen omaa resurssitiedostoa

Toinen testattava web-sovellus on informatiivinen sivusto Tekla.com (https://www.tekla.com). Sivustolta löytyy yleistä informaatiota yrityksen tuotteista, ratkaisuista, ajankohtaisista asioista, tukipalveluista ja yrityksestä itsestään. Sivustoa ylläpidetään sisällönhallintajärjestelmää hyödyntäen ja testeissä keskitytään erityisesti siihen, että sivustonhallintajärjestelmässä luodut sivut ovat saatavilla ja niissä määritetyt tiedot näkyvät oikein. Tähän työhön otettiin tarkastelun alle yksi testi, jossa luodaan sivusto monella eri kappaleella ja sisältötyypillä, minkä jälkeen tarkistetaan sisällön oikeellisuus sivustolla.

Testien rakenne poikkeaa merkittävästi verrattuna Warehousen testeihin. Testien käyttämät avainsanat löytyvät lähes kaikki sekä sovelluksen omasta resurssitiedostosta sekä projektien yhteisestä resurssitiedostosta. Testitapaukset kutsuvat ainoastaan yhtä resurssitiedoston avainsanaa, joka puolestaan käyttää testisuorituksen aikana muita resurssitiedostoon määriteltyjä avainsanoja. Lisäksi sivujen sisällönhallinnassa käytettävä testidata on määritelty erikseen omassa kansiossa oleviin CSV-tiedostoihin. Kuvassa 9 nähdään testidatan rakenne kansiopolussa.

Kuva 9 Tekla.com-testidatan rakenne kansiopolussa

```
robottests/
├─ Teklacom_browser/
|  ├─ testdata/
|  ├─ E2E content creation and verification.robot
├─ resources/
|  ├─ teklacom_browser.resource
|  ├─ common_browser.resource
├─ variables/
|  ├─ teklacom.variables
```

Korvaamistyö lähti jälleen liikkeelle ottamalla nykyisistä projektiin liittyvistä tiedostoista kopiot. Korvaamistyön helpottamiseksi löysin kustomoidun liitännäiskirjaston Robot Frameworkiin, joka on suunniteltu nimenomaan siirtymiseen SeleniumLibraryn käytöstä Browser libraryyn. Kirjasto koostuu kahdesta eri toiminnallisuudesta. Ensimmäinen toiminnallisuus analysoi testeissä käytettäviä SeleniumLibraryn avainsanoja ja tulostaa raportissaan listauksen kaikista SeleniumLibraryn avainsanoista, joita kyseisessä testissä käytetään. Toinen toiminnallisuus mahdollistaa Browser libraryn käytön SeleniumLibraryn avainsanoilla. Browser libraryn toiminnallisuus on siis rakennettu SeleniumLibraryn avainsanojen alle.

Lyhyen käyttökokeilun jälkeen päätin hyödyntää kirjaston ominaisuuksista ainoastaan avainsanojen statistiikkaa. Testiraportissa ei näkynyt jokaisen avainsanan kohdalla Browser libraryn avainsanaa, jota kyseissä avainsanan toiminnossa käytetään. Sen vuoksi päätin luoda oman resurssitiedoston, johon tulisin lisäämään statistiikan perusteella testeissä käytettävien SeleniumLibraryn avainsanojen nimet ja nimien alle Browser libraryn avainsanat, jotka vastaavat samaa toiminnallisuutta. Tällä tyylillä raportistakin pystytään näkemään, mitä Browser libraryn avainsanaa todellisuudessa käytetään. Ohjelmakoodissa 5 nähdään raporttinäkymä sekä korvaavan avainsanan toteutus resurssitiedostossa.

Ohjelmakoodi 5 Raporttinäkymä sekä korvaavan avainsanan toteutus resurssitiedostossa

```
KEYWORD teklacom2021_browser.Navigate to Edit Content page ${type}, ${title}, ${language}
Documentation:        Choose the 1st page in the list with the given type, title & language.
Tags:                 NAVIGATION
Start / End / Elapsed: 20230712 11:05:53.205 / 20230712 11:05:57.337 / 00:00:04.132
− KEYWORD browser_keywords_with_selenium_names.Click Element ${MENU_CONTENT}
  Start / End / Elapsed: 20230712 11:05:53.216 / 20230712 11:05:53.982 / 00:00:00.766
  − KEYWORD Browser.Click ${selector}
    Documentation:        Simulates mouse click on the element found by selector.
    Tags:                 PageContent, Setter
    Start / End / Elapsed: 20230712 11:05:53.221 / 20230712 11:05:53.982 / 00:00:00.761
```

```
Click Element
    [Arguments]    ${selector}
    Browser.Click    ${selector}
```

Kirjaston asennus sujui helposti ja nopeasti. Komennon 2 mukaisella komentokehotteen komennolla kirjasto asentui koneelle pip-järjestelmää hyödyntäen. Kirjasto vaatii

toimiakseen myös Browser libraryn asennuksen, joten se tulee suorittaa tarvittaessa ennen käyttöönottoa.

Komento 2 robotframework-browser-migration asennus

```
pip install robotframework-browser-migration
```

SeleniumLibrary-statistiikan luominen vaatii suoritetun testin **output.xml**-tiedoston. Näin ensimmäinen vaihe statistiikan luomisessa oli suorittaa työssä korvattava testi omalla koneella. Kun testin suoritus päättyi, otin talteen raportista syntyneen **output.xml**-tiedoston sijainnin koneella. Tämän jälkeen avasin komentokehotteen ja suoritin ohjeiden mukaan komennon, jolla statistiikan sai luotua. Kuvassa 10 näkyy sekä suoritettu komento että tuloksena syntynyt taulukko testissä käytetyistä SeleniumLibraryn avainsanoista.

Kuva 10 SeleniumLibrary-avainsanojen statistiikka

```
Command Prompt
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

C:\Users\npolvi>python -m SeleniumStats C:\Users\npolvi\AppData\Local\Temp\RIDEisc86ygu.d\output.xml
reading results from: C:\Users\npolvi\AppData\Local\Temp\RIDEisc86ygu.d\output.xml
+------------------------------------+-------+---------+------------------+
| Keyword                            | count | parents | migration status |
+------------------------------------+-------+---------+------------------+
| Add Cookie                         | 2     | 1       | missing          |
| Capture Page Screenshot            | 121   | 3       |                  |
| Checkbox Should Be Selected        | 1     | 1       |                  |
| Choose File                        | 2     | 1       |                  |
| Click Element                      | 100   | 20      |                  |
| Click Link                         | 11    | 1       |                  |
| Close All Browsers                 | 1     | 1       |                  |
| Create Webdriver                   | 2     | 1       | missing          |
| Delete All Cookies                 | 1     | 1       |                  |
| Element Attribute Value Should Be  | 1     | 1       |                  |
| Execute Javascript                 | 74    | 6       |                  |
| Get Cookie                         | 3     | 1       |                  |
| Get Element Attribute              | 116   | 8       |                  |
| Get Element Count                  | 13    | 7       |                  |
| Get List Items                     | 10    | 1       |                  |
| Get Location                       | 15    | 3       |                  |
| Get Selected List Label            | 22    | 1       |                  |
| Get Text                           | 10    | 4       |                  |
| Get Title                          | 4     | 1       |                  |
| Get Vertical Position              | 45    | 2       |                  |
| Get WebElement                     | 2     | 1       |                  |
| Get WebElements                    | 4     | 1       |                  |
| Go To                              | 6     | 3       |                  |
| Input Text                         | 24    | 3       |                  |
| Location Should Contain            | 4     | 1       |                  |
| Maximize Browser Window            | 2     | 1       |                  |
| Mouse Over                         | 14    | 3       |                  |
| Open Browser                       | 4     | 2       |                  |
| Page Should Contain Element        | 31    | 5       |                  |
| Page Should Not Contain Element    | 6     | 2       |                  |
| Press Keys                         | 4     | 2       | missing          |
| Register Keyword To Run On Failure | 1     | 1       | missing          |
| Reload Page                        | 4     | 2       |                  |
| Scroll Element Into View           | 30    | 2       |                  |
| Select Checkbox                    | 3     | 2       |                  |
| Select From List By Label          | 20    | 1       |                  |
| Set Focus To Element               | 4     | 1       |                  |
| Set Selenium Speed                 | 2     | 1       | missing          |
| Set Window Size                    | 2     | 1       |                  |
| Switch Browser                     | 17    | 4       |                  |
| Wait For Condition                 | 1     | 1       | missing          |
| Wait Until Element Is Enabled      | 12    | 4       |                  |
| Wait Until Element Is Not Visible  | 4     | 2       |                  |
| Wait Until Element Is Visible      | 8     | 2       |                  |
| Wait Until Page Contains           | 20    | 12      |                  |
| Wait Until Page Contains Element   | 149   | 9       |                  |
+------------------------------------+-------+---------+------------------+
```

Avainsanat tulostuvat taulukkomuotoon ja siinä näkyy SeleniumLibrary-avainsanan nimi, käyttökertojen määrä testissä sekä tieto siitä, onko avainsanan toiminnallisuus korvattu kirjastossa. Otin talteen jokaisen avainsanan taulukosta ja loin uuden resurssitiedoston nimeltään **browser_keywords_with_selenium_names.resource**. Tiedoston alkuun lisäsin **Settings**-osioon ohjelmakoodin 6 mukaisen rivin, joka tuo Browser library -kirjaston käyttöön tiedoston sisällä. Lisäksi määrittelin myös timeout-asetuksen kirjaston käytölle, koska

kirjaston oletuksena määritelty 10 sekunnin viive avainsanojen suoritusten onnistumiselle oli monessa kohdassa liian vähän.

Ohjelmakoodi 6 Kirjaston määritys tiedoston sisällä

```
browser_keywords_with_selenium_names.resource ×
resources > browser_keywords_with_selenium_names.resource > ...
      Load in Interactive Console
  1   *** Settings ***
  2   Library    Browser    timeout=0:00:20    strict=False
```

Myös strict-asetukseen täytyi tehdä muutos määrittelemällä sen arvoksi False. Tämä koski etenkin elementtien tunnistamista sivulla, koska muutamassa kohdassa testin suorituksen aikana sivustolla haettava elementti tai teksti esiintyi useamman kerran. SeleniumLibrary-testeissä tämä menee oletuksena läpi, mutta Browser libraryn oletusasetusten mukaan sivustolta saa löytyä ainoastaan yksi elementti, jota ollaan käsittelemässä. Strict-asetuksen määrityksellä tämä voidaan kuitenkin muuttaa, jolloin arvon ollessa False testin suoritus ei epäonnistu, vaikka elementtejä löytyisi useampikin.

Ainoa ongelmatilanne syntyi kustomoidun avainsanan luonnissa, jossa täytyi tehdä määritykset selaimien käynnistämiselle. SeleniumLibraryä käyttävissä testeissä luodaan omat selaimet sekä sivuston luomisessa backend-puolella että sivuston sisällön verifioimisessa frontend-puolella. Näissä testeissä selaimen käynnistämisen yhteydessä voidaan selaimelle määritellä suoraan alias-nimi, jota voidaan hyödyntää myöhemmin selaimien käsittelyn yhteydessä. Browser libraryssä ei ole käytössä aliasta, vaan Browserille, Contextille ja Pagelle muodostuu automaattisesti yksilöivä id-tunniste, jota niiden käsittelyssä tulee käyttää.

Browser library -testieissä toteutin selaimen käynnistämisen niin, että yhden selaininstanssin sisällä luodaan omat Contextit sekä backend- että frontend-sivuille. Näiden contextien id-tunnisteet tallennetaan niiden luomisen yhteydessä omiin muuttujiin ja määritellään testisarjan yleisiksi muuttujiksi, joita käytetään sivujen hallinnoimisen yhteydessä. Erot kirjastojen välisissä toteutuksissa nähdään ohjelmakoodissa 7.

Ohjelmakoodi 7 Selaimen vaihto SeleniumLibrary-testeissä sekä Contextin vaihto Browser library-testeissä

Selaimen vaihto SeleniumLibrary-testeissä

```
IF    ${SUITE_BROWSER_HAS_BEEN_SETUP}
    Switch browser     Frontend
    go to      ${TEKLACOM2021_URL}
ELSE    # No browser
    Browser startup    ${TEKLACOM2021_URL}    alias=Frontend
END
```

Contextin vaihto Browser library -testeissä

```
IF    ${SUITE_BROWSER_HAS_BEEN_SETUP}
    Switch Context    ${FRONTEND_CONTEXT_ID}
    go to     ${TEKLACOM2021_URL}
ELSE    # No browser
    ${FRONTEND_CONTEXT_ID}    common_browser.Browser startup    ${TEKLACOM2021_URL}
    SET SUITE VARIABLE    ${FRONTEND_CONTEXT_ID}
END
```

Contextin id:n tallennus ja arvon palautus avainsanan lopussa

```
New Browser   channel=chrome    headless=false    args=["--start-maximized"]
${context_id}    New Context    viewport=None
New Page    url=${url}
Return From Keyword    ${context_id}
```

Lähes jokaiselle SeleniumLibraryn avainsanalle löytyi vastine Browser librarystä. Tämä korvaustyyli tuntuikin paljon helpommalta ja nopeammalta vaihtoehdolta etenkin, kun avainsanojen toiminnallisuudet olivat tulleet aikaisemmassa korvaustyylissä jo tutuiksi. Ainoat avainsanat, joille ei löytynyt suoraa vastinetta, olivat **Create Webdriver** ja **Set Selenium Speed**. Selainten muodostaminen tapahtuu kuitenkin eri tavalla kirjastojen välillä, joten **Create Webdriver** -avainsanalle ei tarvinnutkaan löytää vastinetta. **Set Selenium Speed** määrittää viiveajan, mikä jokaisen SeleniumLibrary-avainsanan suorituksen jälkeen odotetaan, joten sen vuoksi sillekään ei löydy suoraa vastinetta Browser libraryn avainsanoista. Browser libraryä käytettäessä odotusaika määriteltiin tässä tapauksessa suoraan kirjaston tuomisen yhteydessä.

Kun jokaiselle avainsanalle oli luotu oma kustomoitu avainsana resurssitiedostoon, täytyi testien sisällä muuttaa vielä viittaukset oikeaan kirjastoon. Käytännössä tämä tarkoitti

SeleniumLibrary-kirjaston tuomisen poistamista testitiedostosta ja resurssitiedostoista, jonka jälkeen uuden resurssitiedoston viittaus täytyi lisätä samoihin paikkoihin ohjelmakoodin 8 mukaisesti.

Ohjelmakoodi 8 Resurssitiedoston määritys tiedoston sisällä avainsanojen käyttöönottoon

```
Resource    browser_keywords_with_selenium_names.resource
```

Tällä korvaustyylillä avainsanojen korvaaminen sujui mielestäni erittäin nopeasti ja tehokkaasti. Testissä esiintyvien avainsanojen korvaamiseen kului aikaa selvittelyjen kanssa yhteensä noin kaksi työpäivää, mutta korvaus kattoi suurimman osan projektin oman resurssitiedoston avainsanoista. Mikäli Browser libraryyn tullaan jatkossa siirtymään kokonaan, suurin osa korvaustyöstä on jo tehtynä ja lopullinen siirtyminen tulee sujumaan erittäin nopeasti tämän projektin osalta.

6.3 Testien suoritusaikojen vertailu kirjastojen välillä

Testien korvaamisen jälkeen suoritin vertailua testien suoritusaikojen välillä. Testit suoritettiin tuotantoa vastaavassa kehitysympäristössä. Molempien kirjastojen testit suoritettiin kolme kertaa peräkkäin ja niistä otettiin talteen testiraportit. Näistä testiraporteista kerättiin taulukkoon testeihin kuluneet suoritusajat.

Warehousen testeistä ensimmäisenä vertailussa oli Analytics-testisarja, missä testataan sisällöstä kerättävän analytiikan toimivuus. SeleniumLibraryä käyttävän testisarjan keskiarvo kolmen suorituskerran jälkeen oli 262 sekuntia. Testien välillä ei ollut suurta hajontaa, sillä kaikki suoritusajat olivat kolmen sekunnin sisällä toisistaan. Browser libraryä käyttävän testisarjan keskiarvo oli 187 sekuntia. Hajonta ei ollut suurta myöskään näiden testien välillä, sillä suoritusajat olivat neljän sekunnin sisällä toisistaan.

Seuraava vertailtava testisarja oli Basic View, missä varmistetaan sivuston peruselementtien näkyvyys ja toimivuus. Tämä regressiotestisarja käyttää nykyisissä testiajoissa Firefox-selainta, joten näiden testiajojen kohdalla käytettiin myös Firefoxia. SeleniumLibrary-testien suoritusaika oli keskimäärin 196 sekuntia. Hajonta ei ollut suurta näidenkään testien välillä suoritusaikojen ollessa kahden sekunnin sisällä toisistaan. Browser

library -testien keskimääräinen suoritusaika oli 128 sekuntia. Yhdellä suorituskerralla testien suoritus sujui hieman kahta muuta suorituskertaa pidempään (132 sekuntia).

Kolmas vertailtava testisarja oli Content Translations, missä testaus keskittyy sivulla olevan sisällön käännösteksteihin. SeleniumLibrary-testit antoivat kesimääräiseksi suoritusajaksi 236 sekuntia. Kaikki suoritusajat olivat alle sekunnin sisällä toisistaan. Browser library -testit suoriutuivat jälleen huomattavasti nopeammin keskimääräisen ajan ollessa 117 sekuntia. Hajontaa oli hieman SeleniumLibrary-testejä enemmän, sillä suoritusajat olivat noin kolmen sekunnin sisällä toisistaan.

Neljännessä testisarjassa, Organization Pagessa, testataan organisaation informaatiosivun editointia ja toimivuutta. Vertailu aloitettiin jälleen SeleniumLibrary-testeistä. Keskimääräinen suoritusaika oli 263 sekuntia. Aiemmista sarjoista poiketen hajonta testien suoritusaikojen välillä oli hieman suurempi, sillä testien suoritusajat olivat 10 sekunnin sisällä toisistaan. Browser library -testien keskimääräinen suoritusaika oli 186 sekuntia. Huomionarvoista myös näiden testien kohdalla oli erot suoritusaikojen välillä, sillä testisarja suoriutui lyhyimmillään noin 161 sekunnissa, kun taas pisin suoritusaika oli noin 214 sekuntia.

Copy Content -testisarjassa testaus keskittyy olemassa olevan sisällön kopioimiseen paikasta toiseen. SeleniumLibrary-testit suoriutuivat keskimäärin 131 sekunnissa suoritusaikojen ollessa alle seitsemän sekunnin sisällä toisistaan. Browser library -testit suoriutuivat samasta urakasta keskimäärin 93 sekunnissa. Ero nopeimman ja hitaimman suoritusajan välillä oli noin 12 sekuntia.

Pisin Warehousen testisarja, joka tässä työssä korvattiin, oli Edit Collection and Package. Testisarjassa luodaan ensiksi sisältöä sivustolle, jonka jälkeen testeissä keskitytään sisällön muokkaukseen. SeleniumLibrary-testisarjan keskimääräinen suoritusaika oli 619 sekuntia. Nopein ja hitain suoritusaika olivat noin 10 sekunnin sisällä toisistaan. Browser library -testit suoriutuivat lähes puolet nopeammin keskimääräisen suoritusajan ollessa 316 sekuntia. Ero nopeimman ja hitaimman suoritusajan välillä oli noin 11 sekuntia.

Viimeinen testisarja Warehousessa, joka korvattiin Browser library -kirjaston avainsanoilla, oli Flow-testisarja. Testeissä luodaan aluksi sisältöä, jonka jälkeen sisällön näkyvyys ja

toimivuus testataan kolmen eri käyttäjäprofiilin toimesta. SeleniumLibrary-testien keskimääräinen suoritusaika oli 466 sekuntia. Tämän sarjan kohdalla ero nopeimman ja hitaimman suoritusajan välillä oli noin 18 sekuntia eli hieman aiempia sarjoja suurempi. Browser library -testit olivat jälleen kerran merkittävästi nopeampia keskimääräisen suoritusajan ollessa 277 sekuntia. Hajonta oli tällä kertaa näissä testeissä SeleniumLibrary-testejä pienempi suoritusaikojen ollessa kuuden sekunnin sisällä toisistaan.

Viimeinen vertailu tehtiin Tekla.comin testitapaukselle, jossa sivuston toimivuus varmistetaan päästä päähän -testausmenetelmää noudattaen. Ensiksi sivu ja sinne johdattava navigointilinkki luodaan sisällönhallintajärjestelmän puolella (backend), jonka jälkeen varsinaisella sivustolla navigoidaan juuri luodulle sivulle (frontend) ja vahvistetaan sisällön näkyvyys. SeleniumLibrary-testien keskimääräinen suoritusaika oli 1150 sekuntia. Ero nopeimman ja hitaimman suoritusajan välillä oli peräti 66 sekuntia. Browser library -testien keskimääräinen suoritusaika oli 1080 sekuntia. Myös näiden testien nopeimman ja hitaimman suoritusajan ero oli suuri, noin 69 sekuntia. Kirjastojen välisissä suoritusajoissa ero ei kuitenkaan ollut läheskään yhtä suuri kuin Warehousen testeissä.

Erot testien suoritusajoissa kirjastojen välillä olivat merkittävän suuria Warehousen testeissä. Raportissa ei paljastunut yksittäistä kohtaa, jossa testin suoritus olisi merkittävästi nopeampaa Browser libraryn kanssa, vaan suoritusaika oli tasaisesti nopeampaa testin eri osa-alueilla. Suoritusnopeuden eron huomasi selvästi graafisen käyttöliittymän kautta testejä seuratessa.

Testien luotettavuuden kannalta ei kirjastojen välillä löytynyt juurikaan eroja, kun ajoin testejä omalla koneella. Muutamia epäonnistuneita testiajoja sattui molempien kirjastojen osalta. Nämä liittyivät osin myös suoritusympäristön hetkellisiin muutoksiin. Mahdollisen siirtymisen kohdalla tulee huomioida eri testiympäristöjen sopivuus uuden kirjaston osalta. Tämä osa-alue jää jatkokehityksen varaan. Testien tarkemmat suoritusajat ovat nähtävissä taulukossa 2.

Taulukko 2 Erot testien suoritusajoissa kirjastojen välillä (sekunneissa)

Analytics		
	Browser	**Selenium**
1)	186.038	260.380
2)	185.862	261.008
3)	189.458	263.505
Keskiarvo	**187.119**	**261.631**

Basic View		
	Browser	**Selenium**
1)	125.988	196.253
2)	124.993	194.063
3)	132.112	196.916
Keskiarvo	**127.698**	**195.744**

Content translations		
	Browser	**Selenium**
1)	119.215	236.210
2)	116.146	236.089
3)	117.005	236.539
Keskiarvo	**117.455**	**236.279**

Organization page		
	Browser	**Selenium**
1)	214.370	268.797
2)	161.469	262.162
3)	183.212	257.88
Keskiarvo	**186.350**	**262.946**

Copy Content		
	Browser	**Selenium**
1)	86.358	127.880
2)	98.206	128.156
3)	95.559	135.489
Keskiarvo	**93.374**	**130.508**

Edit collection and package		
	Browser	**Selenium**
1)	310.845	614.307
2)	321.842	624.801
3)	316.248	616.463
Keskiarvo	**316.312**	**618.524**

Flow3		
	Browser	**Selenium**
1)	280.922	463.373
2)	274.589	476.325
3)	274.697	458.546
Keskiarvo	**276.736**	**466.081**

Teklacom E2E content creation and verification		
	Browser	**Selenium**
1)	1121.96	1140.56
2)	1052.86	1187.60
3)	1064.03	1121.57
Keskiarvo	**1079.62**	**1149.91**

7 Johtopäätökset ja pohdinta

Browser libraryn käyttöönotto sekä testien korvaaminen sujui muutamaa ongelmatilannetta lukuun ottamatta hyvin. Jokaiselle SeleniumLibrary-kirjaston avainsanalle löytyi vastine ja työ nopeutui, kun Browser libraryn avainsanojen toiminnallisuuksia oppi tuntemaan paremmin. Näin ollen jokaisen korvattavan avainsanan kohdalla ei tarvinnut työn edetessä enää etsiä dokumentaatiosta vastinetta.

Suoritusaikojen perusteella Browser library on erittäin hyvä vaihtoehto testiautomaation ja web-testauksen toteutukselle projektiemme kohdalla. Testien suoritusajat olivat merkittävästi nopeampia uudella kirjastolla, ja avainsanojen toiminnallisuudet olivat mielestäni suhteellisen helposti opittavissa. Lisäksi käyttöönotto oli suoraviivainen prosessi ja kirjaston ylläpidettävyys on yksinkertaista. Toki Browser library vaatii toimiakseen uuden ohjelman, Node.js:n asennuksen, mutta toisaalta selainajureiden ylläpito hoituu automaattisesti kirjaston puolesta.

Luotettavuudessa ei ollut juurikaan suurta eroa testien onnistumisen kannalta. Molempien kirjastojen testien suorituksissa ilmeni yksittäisiä virhetilanteita, jotka johtuivat hetkellisistä häiriöistä suoritusympäristössä. Avainsanojen korvaaminen tuntui aluksi työläältä projektilta etenkin ensimmäisen tyylin kohdalla, jossa korvasin jokaisen testissä esiintyvän SeleniumLibrary-avainsanan Browser libraryn avainsanalla. Siinä mielessä toinen korvaustyyli, jossa luodaan resurssitiedosto Browser libraryn toiminnallisuuksilla ja SeleniumLibraryn avainsanojen nimillä, oli paljon nopeampi ja suoraviivaisempi tyyli korvata testitapauksia. Ensimmäisessä tyylissä tuli kuitenkin paljon toistoja uusien avainsanojen kohdalla eteen, mikä mielestäni auttoi erittäin paljon sisäistämään uuden kirjaston toiminnallisuuksia.

Browser libraryn käyttöönottokokeilun piiriin ei ollut työn puitteissa mahdollista sisällyttää kaikkia eri toimintoja, joita projektit pitävät sisällään. Näin ollen eteen saattaisi tulla vielä ongelmatilanteita, joissa uuden kirjaston käyttäminen olisi hankalaa. Kirjaston käyttöön liittyvillä keskustelupalstoilla ja tukisivustoilla ei kuitenkaan noussut esille mitään tähän viittaavia asioita, joten pidän sen todennäköisyyttä suhteellisen pienenä.

Yksi merkittävä asia kirjastojen välillä on tukikanavien määrä. SeleniumLibrary on ollut pidempään käytössä ja siihen liittyviä keskustelupalstoja, sivustoja ja muita kanavia on lukematon määrä verkossa. Browser library on suhteellisen uusi kirjasto, että siitä ei löydy vielä niin paljoa materiaalia. Robot Frameworkilla on oma keskustelufoorumi, josta melkein kaikki aiheeseen liittyvät käyttäjien ongelmatilanteet löytyvät. Kirjastolla on omat yhteydenottokanavat, joiden kautta kehittäjille ja käyttäjäkunnalle voi kuitenkin laittaa viestiä mahdollisista haasteista tai virheistä kirjaston käytön parissa. Browser libraryn kehitystyö on myös erittäin aktiivista ja kirjastoon liittyviä päivityksiä tulee lähes kuukausittain.

Kannattaako sitten toimivaa konseptia lähteä rikkomaan? Nykyisissä testisarjoissa ei sinällään ole sellaista ongelmaa, joka pakottaisi harkitsemaan uuden kirjaston käyttöönottoa. On kuitenkin mielestäni tärkeä seurata alalla vaikuttavia trendejä ja kehityksen kulkua. Browser libraryn käyttöönotossa ei paljastunut sellaista ongelmaa, joka estäisi kirjastoon siirtymisen kokonaan. Mikäli erot testien suoritusajoissa olisi muissakin projekteissa yhtä suuria kuin Warehousen testien kohdalla, tulisi pelkästään resurssien osalta huomattavia säästöjä, kun samat testitapaukset pystyttäisiin suorittamaan paljon nopeammin. Lisäksi koodirivien määrä vähenisi, kun ylimääräiset odotukset elementtien ilmaantumiselle saataisiin karsittua pois ja testauksessa pystyttäisiin keskittymään paremmin itse toiminnallisuuksien testaamiseen. Käyttöönotto jää tiimin päätöksen varaan, mutta mielestäni siirtymiselle on hyvät perusteet.

Ensimmäinen tutkimuskysymys koski testiautomaation hyödyntämistä web-sovelluksen kehityksessä. Tämän osalta sain laajasti tuotua testauksen sekä testiautomaation ominaisuuksia esille web-sovelluskehityksen eri vaiheissa. Testiautomaatio kannattaa sisällyttää osaksi kehitystyötä heti alkuvaiheista lähtien, jolloin testiautomaatiota hyödyttävät asiat voidaan ottaa huomioon kehityksen aikana. Web-testaus sisältää myös usein toistuvia vaiheita, jotka ovat otollinen kohde testiautomaation käyttöönotolle.

Toinen tutkimuskysymys käsitteli asioita, joita mahdollisessa kirjaston korvaamisessa tulisi ottaa huomioon. Sain tekstissä tuotua esille kirjastojen toimintaperiaatteet ja rakenteet niiden taustalla. Näin pyrin välittämään lukijalle tietoa siitä, mihin kirjastojen toiminta

perustuu ja miten nämä asiat tulee ottaa huomioon niitä käytettäessä. Kirjastojen toiminta perustuu eri teknologioihin, jonka vuoksi niiden asennus ja ylläpito eroavat toisistaan.

Viimeinen tutkimuskysymys koski ytimekkäästi kannattavuutta uuteen kirjastoon siirtymiselle. Perusteet Browser libraryyn siirtymiselle ovat olemassa, mutta lopullisen päätöksen taustalla vaikuttavat käytössä olevat resurssit ja järkevyys lähteä muuttamaan sinällään toimivaa järjestelmää. Testien suoritusaika oli merkittävästi nopeampi uudella Browser library -kirjastolla, mutta siirtyminen voi kiireisen kehitystyön aikana viedä liikaa aikaa. Uusien kehitysprojektien kohdalla Browser libraryn käyttöönottoa tulee ehdottomasti harkita.

Toimeksiantajan palaute opinnäytetyön osalta oli positiivista. Työstä tuli selkeä paketti, johon saatiin sisällytettyä ennakkoon suunnitellut asiat. Työn toteutuksen aikana pidettiin säännöllisin väliajoin katselmuksia toimeksiantajan ohjaajan kanssa, joissa keskusteltiin työn etenemisestä sekä eteen tulleista haasteista. Kommunikaatio oli avointa koko työn toteutuksen ajan.

8 Yhteenveto

Tutkimuskysymyksiin onnistuin mielestäni vastaamaan hyvin työn puitteissa. Työtä oli helppo lähteä toteuttamaan, kun suunnitteluvaiheessa sain rakennettua järkevän rakenteen työn eri osioille. Luvut kulkevat mielestäni loogisessa järjestyksessä, joka mahdollistaa lukijan syventymisen aiheen pariin. Työn etenemisen seuranta auttoi myös minua pysymään aikataulussa ja sain hyvin opastusta sekä koulun viikoittaisissa katselmuksissa että myös työpaikan ohjaajan kanssa pidetyissä katselmuksissa.

Opin prosessin aikana paljon uutta testauksen parista. Varsinkin teorialuvut toivat minulle paljon uutta näkökulmaa testauksen merkityksestä ja roolista sovelluskehityksen parissa, vaikka olen vuoden verran työskennellyt jo testauksen ja testiautomaation parissa.

Kirjastojen taustalla olevat teknologiat olivat termeinä minulle tuttuja, mutta niiden toimintaperiaatteet ovat olleet hämärän peitossa. Browser library sekä sen taustalla olevat Playwright ja Node.js olivat vieraita ohjelmia minulle ennen työn toteutusta, mutta etenkin Browser libraryn ja Playwright:n toiminnallisuuksista olen saanut paljon uutta oppia. Työn käytännön osuus opetti paljon Browser libraryn käytöstä.

Käytännön osuuden tavoitteet täyttyivät työn osalta hyvin. Sain luotua eri asennusvaiheista dokumentaation, jonka perusteella testausympäristön ja Browser libraryn käyttöönoton pitäisi onnistua ilman ongelmia. Testisarjoja sain korvattua riittävän suuren määrän, jonka perusteella pystyin suorittamaan vertailua testisarjojen suoritusaikojen sekä luotettavuuden välillä.

Mikäli kirjaston siirtymiseen ei päädytä nykyisten projektien kohdalla, tulee Browser libraryn kehitystä kuitenkin seurata tarkalla silmällä. Kirjaston kehityksen taustalla on iso joukko kehittäjiä ja päivityksiä tulee useaan otteeseen. Robot Frameworkiin liittyvissä konferensseissa on ollut monia esityksiä Browser libraryn tuomista mahdollisuuksista ja hyödyistä web-testauksen parissa.

Lähteet

A Beginner's Guide to Automated Website Testing. (ei pvm.). Noudettu 14. huhtikuuta 2023, osoitteesta https://testsigma.com/website-testing

Automated Smoke Testing: Everything You Need to Know. (ei pvm.). *Rainforest QA*. Noudettu 15. huhtikuuta 2023, osoitteesta https://www.rainforestqa.com/blog/automated-smoke-testing

Browser Library. (ei pvm.). Noudettu 24. maaliskuuta 2023, osoitteesta https://robotframework-browser.org/

Calvello, M. (2022, marraskuuta 8). *The 4 Levels of Testing in Software Engineering Explained | Fellow*. Fellow.App. https://fellow.app/blog/engineering/the-levels-of-testing-in-software-engineering-explained/

Cavero-Baptista, L. (ei pvm.). *Web Application Testing: The Basics of Web App Test Automation*. Noudettu 13. huhtikuuta 2023, osoitteesta https://www.leapwork.com/blog/web-application-testing-the-basics-of-web-app-test-automation

Debugging Tests | Playwright. (ei pvm.). Noudettu 27. huhtikuuta 2023, osoitteesta https://playwright.dev/docs/debug

Doğan, S., Betin-Can, A., & Garousi, V. (2014). Web application testing: A systematic literature review. *Journal of Systems and Software*, *91*, 174–201. https://doi.org/10.1016/j.jss.2014.01.010

Fast and reliable end-to-end testing for modern web apps | Playwright. (ei pvm.). Noudettu 24. maaliskuuta 2023, osoitteesta https://playwright.dev/

Garousi, V., & Elberzhager, F. (2017). Test Automation: Not Just for Test Execution. *IEEE Software*, *34*(2), 90–96. https://doi.org/10.1109/MS.2017.34

Hamilton, T. (2020a, tammikuuta 2). *What is Software Testing? Definition*. https://www.guru99.com/software-testing-introduction-importance.html

Hamilton, T. (2020b, helmikuuta 6). *Web Application Testing: How to Test a Website?* https://www.guru99.com/web-application-testing.html

Homann, M. (ei pvm.-a). *What is Selenium Testing?* Noudettu 22. huhtikuuta 2023, osoitteesta https://www.leapwork.com/blog/what-is-selenium-testing

Homann, M. (ei pvm.-b). *Why Automate Regression Testing? 5 Reasons*. Noudettu 11. kesäkuuta 2023, osoitteesta https://www.leapwork.com/blog/5-reasons-why-you-should-automate-regression-testing

Homès, B. (2012). *Fundamentals of Software Testing*. John Wiley & Sons, Incorporated. http://ebookcentral.proquest.com/lib/hamk-ebooks/detail.action?docID=1120766

How to write good test case. (2023). [Software]. Robot Framework. https://github.com/robotframework/HowToWriteGoodTestCases/blob/694b2d666048093e294ef123d56c52a9defb4a1a/HowToWriteGoodTestCases.rst (Original work published 2016)

ISTQB. (2018). *Sertifioitu testaaja. Perustason sertifikaattisisältö (käännösversio 10.10.2018, alkuperäisversio 4.6.2018)*.

Joe. (2022, elokuuta 15). *Microsoft Playwright Testing Guide and Tutorial | Test Guild*. Automation Testing Made Easy Tools Tips & Training. https://testguild.com/what-is-microsoft-playwright-js/

Juvonen, R. (2018). *Ohjelmistoprojektin sudenkuopat ja miten ne vältetään*. https://www.ellibslibrary.com/book/9789528001454/ohjelmistoprojektin-sudenkuopat-ja-miten-ne-valtetaan

Kasurinen, J. P. (2014). *Ohjelmistotestauksen käsikirja*. Docendo. https://www.ellibslibrary.com/book/978-952-5912-99-9/ohjelmistotestauksen-kasikirja

Knight, A. (2022, maaliskuuta 31). *Testing Storybook Components in Any Browser – Without Writing Any New Tests!* Automated Visual Testing | Applitools. https://applitools.com/blog/storybook-components-cross-browser-testing/

Knowit (Ohjaaja). (2020, huhtikuuta 7). *Robot Framework Tutorial 1 – Johdanto*. https://www.youtube.com/watch?v=H9YVlFKdOeM

Ohjelmistotestaus ja laadunvarmistus. (ei pvm.). VALA. Noudettu 24. maaliskuuta 2023, osoitteesta https://www.valagroup.com/fi/palvelut/ohjelmiston-laadunvarmistus/ohjelmistotestaus-ja-laadunvarmistus/

Pandey, S. (2023, tammikuuta 25). Playwright vs Selenium—Which is Faster and More Stable? *Version 1*. https://medium.com/version-1/playwright-vs-selenium-which-is-faster-and-more-stable-c41ba1a89c0a

Peres, H. (2018). *Automating Software Tests Using Selenium*. Scribl. http://ebookcentral.proquest.com/lib/hamk-ebooks/detail.action?docID=6375573

Robot Framework. (ei pvm.). Noudettu 24. maaliskuuta 2023, osoitteesta https://robotframework.org/

Robot Framework User Guide. (ei pvm.). Noudettu 29. huhtikuuta 2023, osoitteesta https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Robotframework-browser. (2023). [Python]. marketsquare. https://github.com/MarketSquare/robotframework-browser (Original work published 2020)

Schmitt, J. (2022, huhtikuuta 4). *What is end-to-end testing?* CircleCI. https://circleci.com/blog/what-is-end-to-end-testing/

Selenium Overview. (ei pvm.). Selenium. Noudettu 24. maaliskuuta 2023, osoitteesta https://www.selenium.dev/documentation/overview/

SeleniumLibrary. (ei pvm.). Noudettu 24. maaliskuuta 2023, osoitteesta https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html

Software Testing | Basics. (2017, elokuuta 8). *GeeksforGeeks*. https://www.geeksforgeeks.org/software-testing-basics/

Testausautomaatio tehostaa laadunvarmistusta—Blogi | ATR Soft. (2018, helmikuuta 12). https://www.atrsoft.com/blogi/teknologiat/testausautomaatio-tehostaa-laadunvarmistusta

Testiautomaatio varmistaa onnistuneen kehittämisen—Reflector Oy. (ei pvm.). Noudettu 13. heinäkuuta 2023, osoitteesta https://reflector.fi/testiautomaatio-varmistaa-onnistuneen-kehittamisen/

Tihekari, T. (2022, huhtikuuta 4). *Miksi E2E-testiautomaatiota kannattaa hyödyntää?* City Dev Labs. https://citydevlabs.fi/miksi-e2e-testiautomaatiota-kannattaa-hyodyntaa/

Trace viewer | Playwright. (ei pvm.). Noudettu 28. huhtikuuta 2023, osoitteesta https://playwright.dev/docs/trace-viewer-intro

Vijay. (2023, maaliskuuta 18). *Web Application Testing Guide: How To Test A Website*. Software Testing Help. https://www.softwaretestinghelp.com/web-application-testing/

What Exactly is Node.js? Explained for Beginners. (2022, joulukuuta 5). FreeCodeCamp.Org. https://www.freecodecamp.org/news/what-is-node-js/

What Is Node.js and Why You Should Use It. (2023, maaliskuuta 9). Kinsta®. https://kinsta.com/knowledgebase/what-is-node-js/

What Is Playwright – A Tutorial on How to Use Playwright. (ei pvm.). Noudettu 23. huhtikuuta 2023, osoitteesta https://www.lambdatest.com/playwright

What is Robot Framework? The Story Behind a Robotic Testing Ecosystem. (2022, tammikuuta 6). https://www.copado.com/devops-hub/blog/what-is-robot-framework-the-story-behind-a-robotic-testing-ecosystem-crt

Liite 1: Aineistonhallintasuunnitelma

Kehitysprojektin aikana otetaan kuvakaappauksia käyttöönoton eri vaiheissa. Lisäksi havaintoja kirjoitetaan ylös projektin edetessä. Nämä tiedot analysoidaan opinnäytetyön tuloksissa ja pohdinnassa.

Sovelluksen ja kirjaston asennus tehdään kotikoneelle (asennusopas) sekä työpaikan ympäristöön. Työpaikan ympäristössä data säilytetään omalla levyasemalla sekä varmuuskopiointi tehdään säännöllisesti Googlen pilvipalveluun. Kotikoneen data säilytetään omalla levyasemalla. Varmuuskopio tehdään säännöllisin väliajoin Microsoftin pilvipalveluun sekä erilliselle muistitikulle. Molempien ympäristöjen dataa säilytetään omilla levyasemillaan vuoden ajan työn hyväksymisestä.

Käytännön osuutta varten luodaan oma haara testien versionhallintaan. Näin vältetään sekaannukset nykyisten testitapausten kanssa ja mahdollistetaan uusien testitapausten säilöminen muualla kuin omalla tietokoneella. Työpaikan ympäristön datankeräyksessä kiinnitetään huomiota siihen, ettei työhön päädy järjestelmää koskevaa kriittistä informaatiota.

Työn kirjoitus tapahtuu kotikoneella Microsoftin ohjelmistolla, joten työhön kerättävä data tulee siirtää raportointivaiheessa kotikoneen ympäristöön. Tämä tehdään hyödyntäen Googlen pilvipalvelua.

Opinnäytetyöaineiston jatkokäyttö työn valmistumisen jälkeen

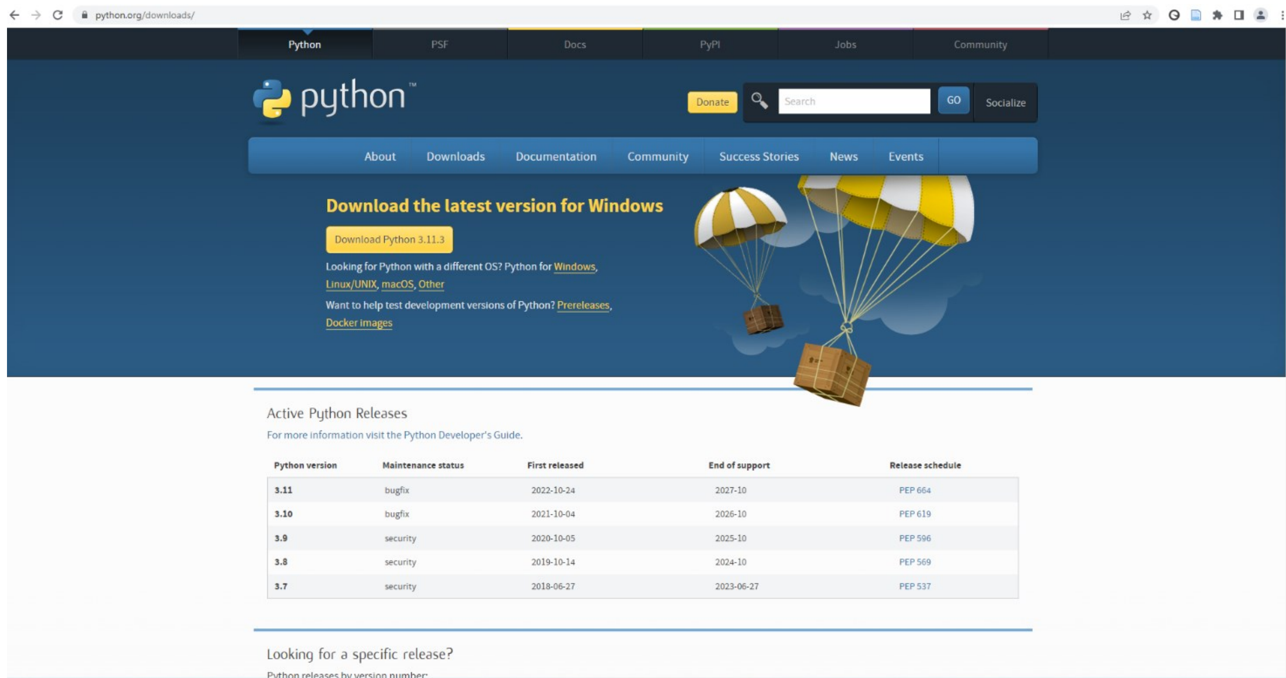
1. Et halua hyödyntää tai antaa tutkimusaineistoasi jatkokäyttöön

Tutkimusaineistoa ei jatkokäytetä. Opinnäytetyön tekijä säilyttää aineiston tietoturvallisesti vuoden ajan opinnäytetyön hyväksymispäivästä, jotta opinnäytetyön tulokset voidaan tarvittaessa varmistaa ja hävittää tämän jälkeen aineiston tietoturvallisesti.

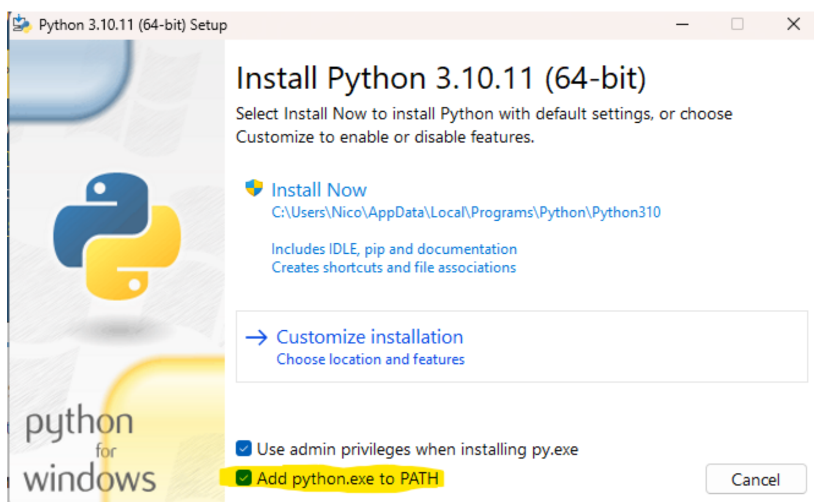
Liite 2: Asennusdokumentaatio

Python

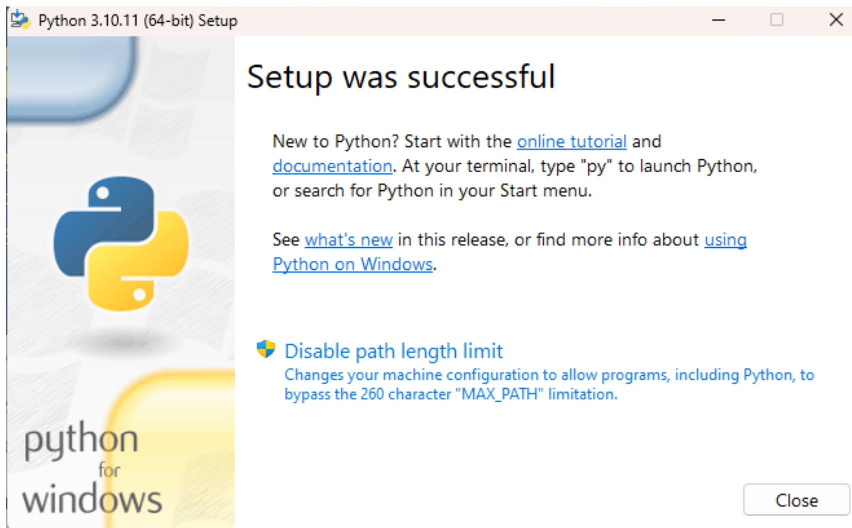
Pythonin asennustiedosto kannattaa hakea suoraan Pythonin omilta verkkosivuilta (https://python.org/downloads). Tässä ohjeessa asennamme uusimman version Windows-käyttöjärjestelmälle lataamalla asennustiedoston keltaisesta latauspainikkeesta



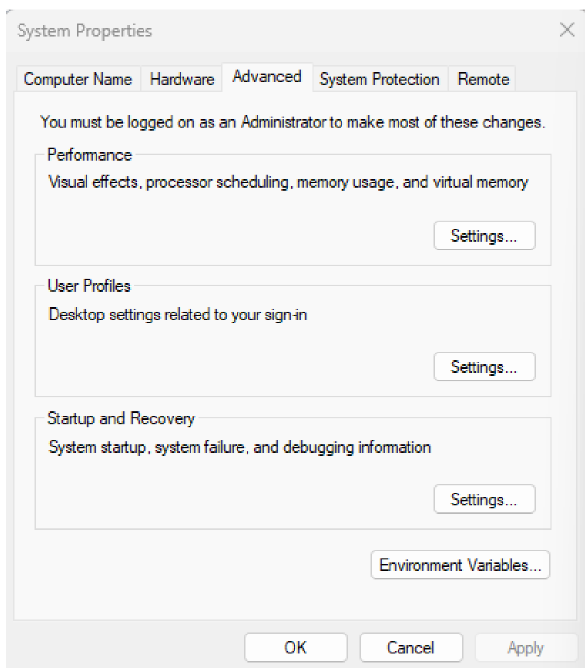
Latauksen jälkeen käynnistetään asennusohjelma. Asennuksen voi suorittaa ns. pika-asennuksena tai mukautetusti, jolloin käyttäjä voi valita mm. asennuspolun sekä muokata asennettavia ominaisuuksia. Käytetään tässä tapauksessa pika-asennusta ja määritetään ennen asennuksen aloitusta ”Add python.exe to PATH”, jolloin python lisätään automaattisesti asennuksen yhteydessä koneen ympäristömuuttujan polkuun. Asennus aloitetaan klikkaamalla ”Install now”.



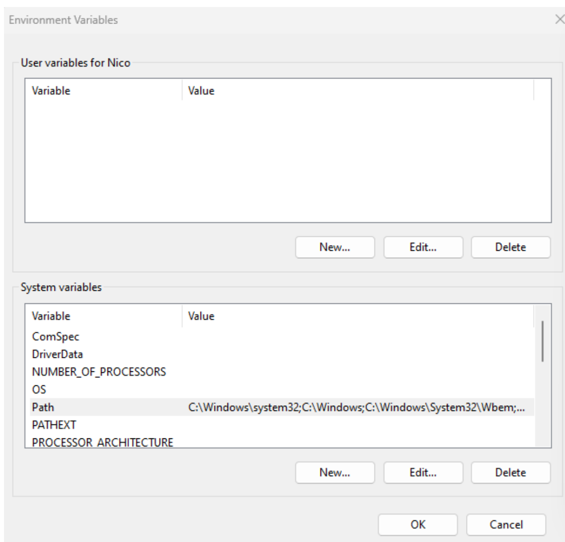
Asennus on valmis. Tässä kohtaa käyttäjä voi muokata polun enimmäispituutta (260 merkkiä), jolloin Pythonin asennuskansio varmasti pystytään lisäämään polkuun.



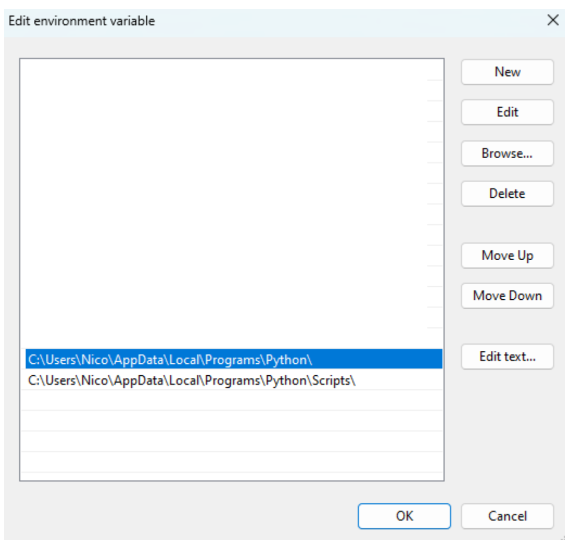
Polun voi käydä tarkistamassa käsin koneen ympäristömuuttujista. Käytössäni on Windowsin englanninkielinen versio, jolloin ympäristömuuttujat löytyvät seuraavalla tavalla: Kirjoitetaan Windowsin hakukenttään ”Edit environment variables” ja valitaan ”Edit the system environment variables”. Avautuvasta System Properties-valikosta valitaan ”Environment Variables…”



Ympäristömuuttujat koostuvat kahdesta osasta: käyttäjän omista muuttujista (User variables) sekä järjestelmämuuttujista (System variables). Valitaan System variables -kohdasta ”Path” ja klikataan ”Edit…”



Polusta pitäisi löytyä sekä Pythonin asennuskansio että sen alla oleva Scripts-kansio. Mikäli asennusvaiheessa käyttäjä ei valinnut polun lisäystä tai muusta syystä polkua ei löydy, se voidaan lisätä ”New”-painikkeesta. Mikäli asennuskansio on väärä (vanha asennuspaikka, monia versioita yms.), polun voi muokata valitsemalla Python-kansion rivin ja painamalla ”Edit”.



Pythonin asennuksen voi todeta onnistuneeksi kirjoittamalla komentokehotteeseen ”python –version”. Vastauksena on pythonin versio, joka löytyy Path-määrittelyn mukaisesta kansiosta.

```
Command Prompt
Microsoft Windows [Version 10.0.22621.1702]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nico>python --version
Python 3.10.11

C:\Users\Nico>
```

Robot Framework

Pythonin asennuksen jälkeen voidaan siirtyä Robot Frameworkin asennukseen. Tämä onnistuu helpoiten kirjoittamalla komentokehotteeseen ”pip install robotframework”, jolloin asennus käynnistyy. Asennuksen aikana käyttäjän ei tarvitse tehdä muuta kuin odottaa asennuksen valmistumista. Kun asennus on onnistunut, komentokehotteeseen tulee ilmoitus ”Successfully installed robotframework...”.

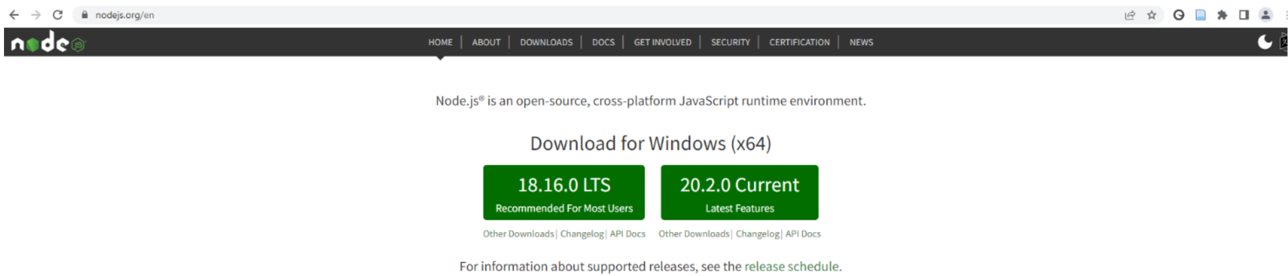
```
C:\Users\Nico>pip install robotframework
Collecting robotframework
  Downloading robotframework-6.0.2-py3-none-any.whl (658 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 658.7/658.7 kB 10.5 MB/s eta 0:00:00
Installing collected packages: robotframework
Successfully installed robotframework-6.0.2
```

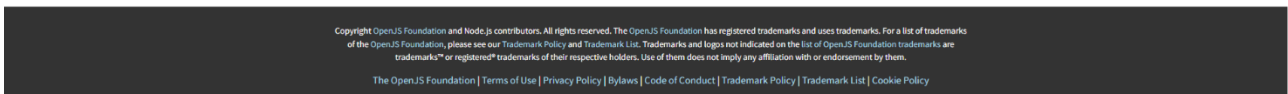
Komentokehotteen kautta voi myös tarkistaa Robot Frameworkin versionumeron komennolla ”robot –version”. Lisäksi komento näyttää Pythonin versionumeron, jota Robot Framework käyttää.

```
Command Prompt

Microsoft Windows [Version 10.0.22621.1702]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nico>robot --version
Robot Framework 6.0.2 (Python 3.10.11 on win32)

C:\Users\Nico>
```

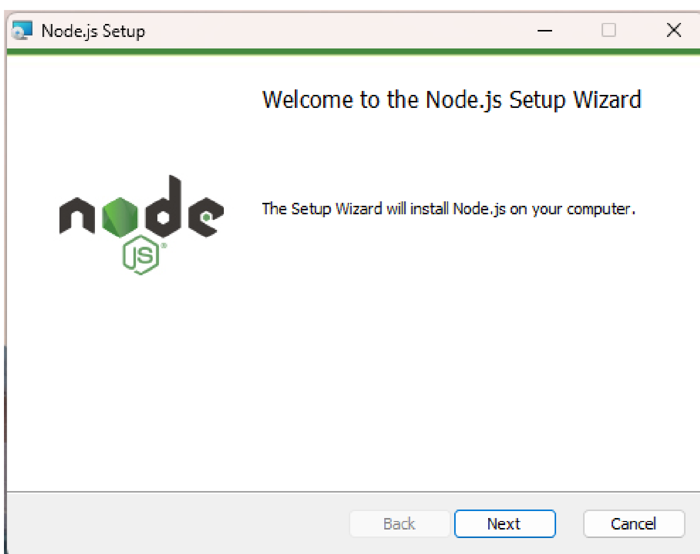
Node.js

Node.js kannattaa Pythonin tapaan ladata suoraan ohjelman omilta verkkosivuilta (https://nodejs.org). Etusivulta löytyy suoraan latauslinkit uusimmalle versiolle, joka sisältää uusimmat ominaisuudet ja päivitykset. Toinen linkki on versiolle, joka on suositus suurimalle osalle käyttäjistä. Se ei sisällä vielä uusimpia päivityksiä, mutta se sisältää todennäköisesti vähemmän virheitä. Node.js:ää emme itsessään käytä testien kehityksessä, vaan se toimii Browser libraryn taustalla, joten versiovalinnalla ei tässä kohtaa ole suurempaa merkitystä. Valitaan versio, joka on suositus useimmille käyttäjille.

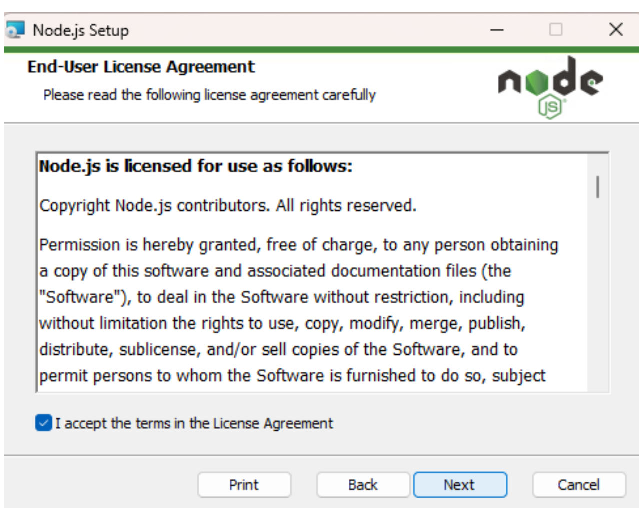




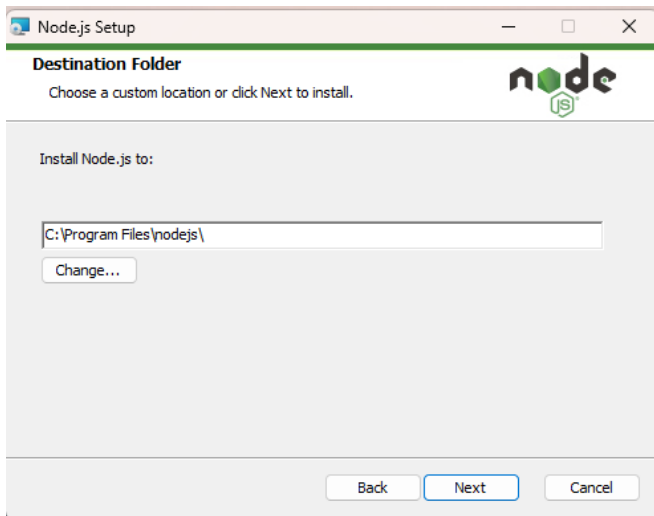
Latauksen jälkeen avataan asennustiedosto ja aloitetaan asennus.



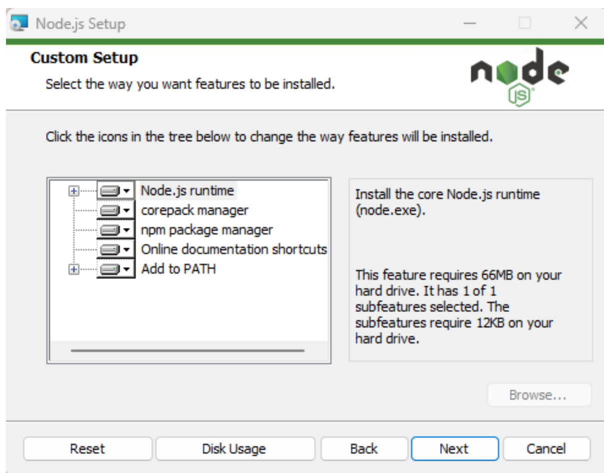
Hyväksytään lisenssiehdot



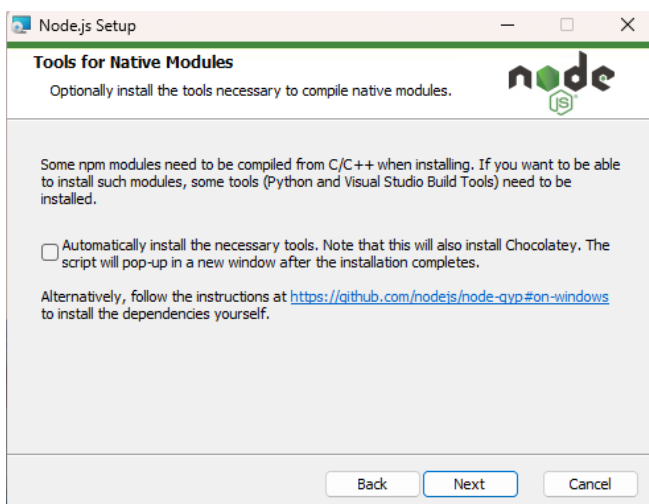
Seuraavassa vaiheessa käyttäjä voi määrittää asennuskansion paikan. Tässä tapauksessa jätän oletusvalinnan.



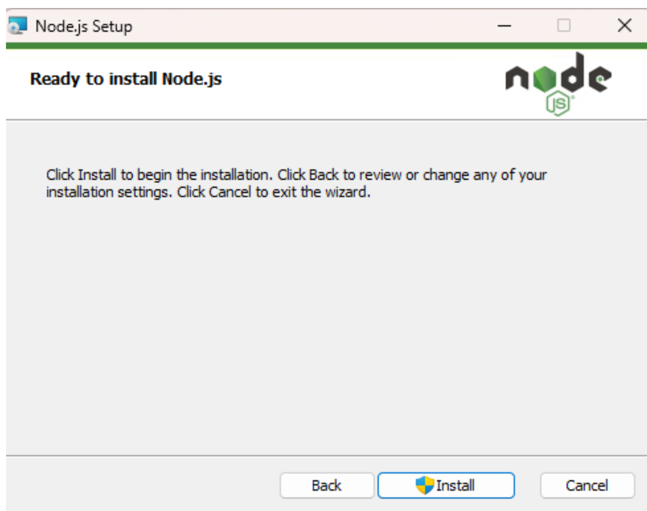
Seuraavaksi käyttäjä voi kustomoida asennusta halutessaan. Ei muuteta valintoja, vaan mennään oletusasetuksilla, jolloin asennus lisää myös Node.js:n järjestelmämuuttujan polkuun (Path).



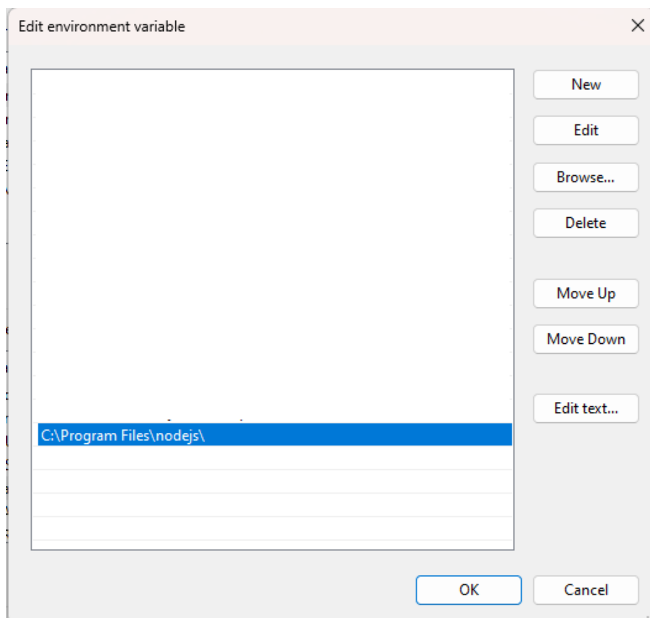
Käyttäjä voi halutessaan asentaa myös lisäosia ja työkaluja, mutta jätetään tässä tapauksessa valinta tässä kohdassa pois.



Viimeisenä vaiheena suoritetaan asennus.



Käydään vielä tarkastamassa Path ja huomataan, että asennus lisäsi nodejs:n asennuskansion sinne.



Browser library

Browser libraryn asennus suoritetaan yhdellä komentokehotteen komennolla ”pip install robotframework-browser”. Kuvakaappaukset on pilkottu kolmeen osaan, jossa näytetään asennusvaiheet. Käyttäjän ei tarvitse tässäkään vaiheessa tehdä muuta kuin odottaa asennuksen valmistumista.

```
C:\Users\Nico>pip install robotframework-browser
Collecting robotframework-browser
  Downloading robotframework_browser-16.0.2-py3-none-any.whl (266 kB)
     ---------------------------------------- 266.3/266.3 KB 16.0 MB/s eta 0:00:00
Collecting wrapt>=1.14.1
  Downloading wrapt-1.15.0-cp310-cp310-win_amd64.whl (36 kB)
Collecting overrides>=7.3.1
  Downloading overrides-7.3.1-py3-none-any.whl (17 kB)
Collecting grpcio-tools==1.51.3
  Downloading grpcio_tools-1.51.3-cp310-cp310-win_amd64.whl (1.8 MB)
     ---------------------------------------- 1.8/1.8 MB 19.7 MB/s eta 0:00:00
Collecting typing-extensions>=4.4.0
  Downloading typing_extensions-4.5.0-py3-none-any.whl (27 kB)
Collecting protobuf==4.22.1
  Downloading protobuf-4.22.1-cp310-abi3-win_amd64.whl (420 kB)
     ---------------------------------------- 420.6/420.6 KB 27.4 MB/s eta 0:00:00
```

```
Collecting robotframework-pythonlibcore>=4.1.2
  Downloading robotframework_pythonlibcore-4.1.2-py2.py3-none-any.whl (10 kB)
Collecting backports.cached-property>=1.0.2
  Downloading backports.cached_property-1.0.2-py3-none-any.whl (6.1 kB)
Requirement already satisfied: robotframework>=5.0.1 in c:\python\lib\site-packages (from robotframework-browser) (6.0.2)
Collecting grpcio==1.51.3
  Downloading grpcio-1.51.3-cp310-cp310-win_amd64.whl (3.7 MB)
```

```
Collecting robotframework-assertion-engine>=1.0.0
  Downloading robotframework_assertion_engine-1.0.0-py3-none-any.whl (8.5 kB)
Requirement already satisfied: setuptools in c:\python\lib\site-packages (from grpcio-tools==1.51.3->robotframework-browser) (58.1.0)
Installing collected packages: wrapt, typing-extensions, robotframework-pythonlibcore, protobuf, overrides, grpcio, backports.cached-property, robotframework-assertion-engine, grpcio-tools, robotframework-brow
ser
Successfully installed backports.cached-property-1.0.2 grpcio-1.51.3 grpcio-tools-1.51.3 overrides-7.3.1 protobuf-4.22.1 robotframework-assertion-engine-1.0.0 robotframework-browser-16.0.2 robotframework-pytho
nlibcore-4.1.2 typing-extensions-4.5.0 wrapt-1.15.0
WARNING: You are using pip version 22.0.4; however, version 23.0.1 is available.
You should consider upgrading via the 'C:\Python\python.exe -m pip install --upgrade pip' command.
```

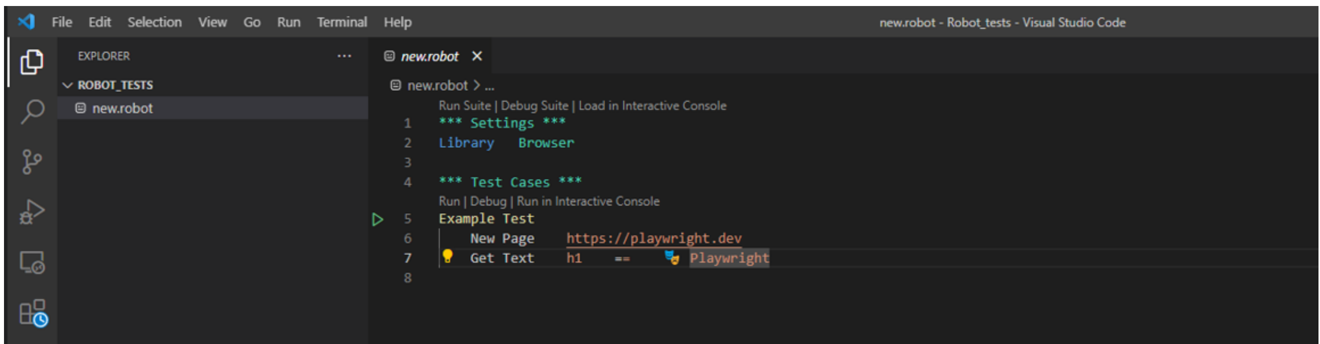
Kun asennus on suoritettu loppuun, kirjasto täytyy vielä alustaa käyttökuntoon komennolla ”rfbrowser init”.

```
C:\Users\Nico>rfbrowser init
2023-04-05 12:24:06,820 [INFO    ] ==================================================================================================
2023-04-05 12:24:06,820 [INFO    ] Installing node dependencies...
2023-04-05 12:24:12,277 [INFO    ] Installing rfbrowser node dependencies at C:\Python\lib\site-packages\Browser\wrapper2023-04-05 12:24:13,876 [INFO    ] npm WARN config production Use `--omit=dev` instead.

2023-04-05 12:24:38,876 [INFO    ]

2023-04-05 12:24:38,876 [INFO    ] added 61 packages, and audited 62 packages in 25s

2023-04-05 12:24:38,876 [INFO    ]

2023-04-05 12:24:38,876 [INFO    ] 5 packages are looking for funding

2023-04-05 12:24:38,876 [INFO    ]   run `npm fund` for details

2023-04-05 12:24:38,876 [INFO    ]

2023-04-05 12:24:38,876 [INFO    ] found 0 vulnerabilities

2023-04-05 12:24:38,876 [INFO    ] npm notice

2023-04-05 12:24:38,876 [INFO    ] npm notice New minor version of npm available! 9.5.0 -> 9.6.3

2023-04-05 12:24:38,876 [INFO    ] npm notice Changelog: <https://github.com/npm/cli/releases/tag/v9.6.3>

2023-04-05 12:24:38,876 [INFO    ] npm notice Run `npm install -g npm@9.6.3` to update!

2023-04-05 12:24:38,876 [INFO    ] npm notice

2023-04-05 12:24:38,876 [INFO    ]
2023-04-05 12:24:38,876 [INFO    ]
2023-04-05 12:24:38,892 [INFO    ] rfbrowser init completed
2023-04-05 12:24:38,892 [INFO    ] ==================================================================================================
```

Tämän jälkeen testien kirjoitus voi alkaa. Alla on vielä esimerkki tilanteesta, jossa kirjasto oli asennettu onnistuneesti. Alustus oli kuitenkin jäänyt vielä tekemättä.

```
File  Edit  Selection  View  Go  Run  Terminal  Help          • new.robot - Robot_tests - Visual Studio Code

EXPLORER                  new.robot 3 •
ROBOT_TESTS               new.robot > ...
  new.robot   3           Run Suite | Debug Suite | Load in Interactive Console
                          1  *** Settings ***
                          2  Library    Browser
                          3
                          4  *** Test C
                             Run | Debug |
                          5  Example Te
                          6      New Pa
                          7      Get Te
                          8

Unresolved library: Browser.
Error generating libspec:
Initializing library 'Browser' with no arguments failed: Could not find node dependencies in installation directory
`C:\Python\lib\site-packages\Browser\wrapper.` Run `rfbrowser init` to install the dependencies. robotframework
View Problem   Quick Fix... (Ctrl+.)
```

Ja tältä editori näyttää alustuksen jälkeen ilman virheilmoituksia:



Kirjasto oli ehtinyt päivittyä työn suoritushetkellä, joten käydään läpi myös päivityksen suorittaminen. Koko päivitysprosessi tapahtuu kolmessa vaiheessa. Ensimmäisenä suoritetaan varsinainen päivitys, jolloin uudet tiedostot ladataan koneelle. Tämä tapahtuu komennolla ”pip install –upgrade robotframework-browser”.

```
C:\Users\Nico>pip install --upgrade robotframework-browser
Requirement already satisfied: robotframework-browser in c:\python\lib\site-packages (16.0.2)
Collecting robotframework-browser
  Downloading robotframework_browser-16.1.0-py3-none-any.whl (267 kB)
     ---------------------------------------- 267.3/267.3 kB 3.3 MB/s eta 0:00:00
Requirement already satisfied: overrides>=7.3.1 in c:\python\lib\site-packages (from robotframework-browser) (7.3.1)
Requirement already satisfied: wrapt>=1.14.1 in c:\python\lib\site-packages (from robotframework-browser) (1.15.0)
Requirement already satisfied: backports.cached-property>=1.0.2 in c:\python\lib\site-packages (from robotframework-browser) (1.0.2)
Collecting protobuf==4.22.4
  Downloading protobuf-4.22.4-cp310-abi3-win_amd64.whl (420 kB)
     ---------------------------------------- 420.6/420.6 kB 27.4 MB/s eta 0:00:00
Collecting grpcio-tools==1.54.0
  Downloading grpcio_tools-1.54.0-cp310-cp310-win_amd64.whl (1.7 MB)
     ---------------------------------------- 1.7/1.7 MB 9.8 MB/s eta 0:00:00
Requirement already satisfied: robotframework-pythonlibcore>=4.1.2 in c:\python\lib\site-packages (from robotframework-browser) (4.1.2)
Requirement already satisfied: robotframework-assertion-engine>=1.0.0 in c:\python\lib\site-packages (from robotframework-browser) (1.0.0)
Collecting grpcio==1.54.0
  Downloading grpcio-1.54.0-cp310-cp310-win_amd64.whl (4.1 MB)
     ---------------------------------------- 4.1/4.1 MB 2.2 MB/s eta 0:00:00
Requirement already satisfied: robotframework>=5.0.1 in c:\python\lib\site-packages (from robotframework-browser) (6.0.2)
Requirement already satisfied: typing-extensions>=4.4.0 in c:\python\lib\site-packages (from robotframework-browser) (4.5.0)
Requirement already satisfied: setuptools in c:\python\lib\site-packages (from grpcio-tools==1.54.0->robotframework-browser) (65.5.0)
Installing collected packages: protobuf, grpcio, grpcio-tools, robotframework-browser
  Attempting uninstall: protobuf
    Found existing installation: protobuf 4.22.1
    Uninstalling protobuf-4.22.1:
      Successfully uninstalled protobuf-4.22.1
  Attempting uninstall: grpcio
    Found existing installation: grpcio 1.51.3
    Uninstalling grpcio-1.51.3:
      Successfully uninstalled grpcio-1.51.3
  Attempting uninstall: grpcio-tools
    Found existing installation: grpcio-tools 1.51.3
    Uninstalling grpcio-tools-1.51.3:
      Successfully uninstalled grpcio-tools-1.51.3
  Attempting uninstall: robotframework-browser
    Found existing installation: robotframework-browser 16.0.2
    Uninstalling robotframework-browser-16.0.2:
      Successfully uninstalled robotframework-browser-16.0.2
Successfully installed grpcio-1.54.0 grpcio-tools-1.54.0 protobuf-4.22.4 robotframework-browser-16.1.0

[notice] A new release of pip is available: 23.0.1 -> 23.1.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Päivityksen jälkeen kirjaston node-riippuvaisuudet pitää poistaa uusien tieltä. Tämä suoritetaan komennolla ”rfbrowser clean-node”. Lopuksi suoritetaan vielä alustus komennolla ”rfbrowser init”.

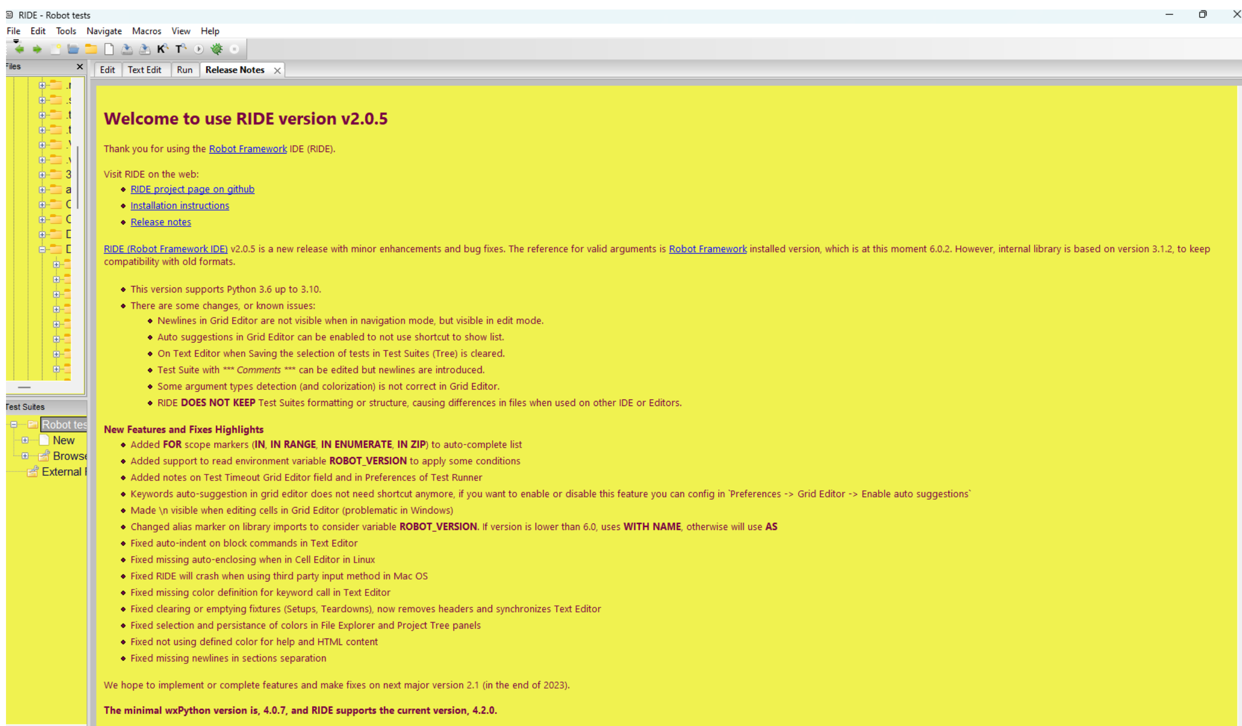
```
C:\Users\Nico>rfbrowser clean-node
2023-05-14 19:32:43,663 [INFO    ] Delete library node dependencies...

C:\Users\Nico>rfbrowser init
2023-05-14 19:32:52,835 [INFO    ] ==============================================================================================================
2023-05-14 19:32:52,835 [INFO    ] Installing node dependencies...
2023-05-14 19:32:58,367 [INFO    ] Installing rfbrowser node dependencies at C:\Python\lib\site-packages\Browser\wrapper
2023-05-14 19:32:59,492 [INFO    ] npm WARN config production Use `--omit=dev` instead.

2023-05-14 19:33:26,743 [INFO    ]

2023-05-14 19:33:26,743 [INFO    ] added 61 packages, and audited 62 packages in 27s

2023-05-14 19:33:26,743 [INFO    ]

2023-05-14 19:33:26,743 [INFO    ] 5 packages are looking for funding

2023-05-14 19:33:26,743 [INFO    ]   run `npm fund` for details

2023-05-14 19:33:26,743 [INFO    ]

2023-05-14 19:33:26,743 [INFO    ] found 0 vulnerabilities

2023-05-14 19:33:26,743 [INFO    ] npm notice

2023-05-14 19:33:26,743 [INFO    ] npm notice New minor version of npm available! 9.5.0 -> 9.6.6

2023-05-14 19:33:26,743 [INFO    ] npm notice Changelog: <https://github.com/npm/cli/releases/tag/v9.6.6>

2023-05-14 19:33:26,743 [INFO    ] npm notice Run `npm install -g npm@9.6.6` to update!

2023-05-14 19:33:26,743 [INFO    ] npm notice

2023-05-14 19:33:26,758 [INFO    ]
2023-05-14 19:33:26,758 [INFO    ]
2023-05-14 19:33:26,758 [INFO    ] rfbrowser init completed
2023-05-14 19:33:26,758 [INFO    ] ==============================================================================================================
```

RIDE ja Robot Framework Language Server-lisäosa Visual Studio Codeen

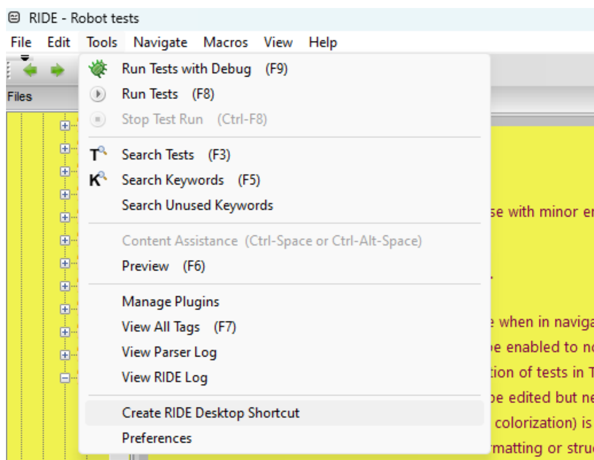
Kun tarvittavat ohjelma-asennukset on saatu suoritettua loppuun, tarvitsemme vielä kehitysympäristön testien tekemistä varten. Ensimmäisenä asennetaan RIDE. Tämän asennus jatkaa Robot Frameworkin ja kirjastojen tapaan suoraviivaista komento-pohjaista asennuslinjaa. RIDEn asennus tapahtuu yhdellä komennolla ”pip install -U robotframework-ride”, jolloin järjestelmä asentaa uusimman versiojulkaisun editorista.

```
C:\Users\Nico>pip install -U robotframework-ride
Collecting robotframework-ride
  Downloading robotframework-ride-2.0.5.tar.gz (1.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.4/1.4 MB 17.7 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Collecting PyPubSub
  Using cached Pypubsub-4.0.3-py3-none-any.whl (61 kB)
Collecting Pygments
  Downloading Pygments-2.15.1-py3-none-any.whl (1.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 18.3 MB/s eta 0:00:00
Collecting psutil
  Downloading psutil-5.9.5-cp36-abi3-win_amd64.whl (255 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 255.1/255.1 kB ? eta 0:00:00
Collecting wxPython
  Using cached wxPython-4.2.0-cp310-cp310-win_amd64.whl (18.0 MB)
Collecting Pywin32
  Downloading pywin32-306-cp310-cp310-win_amd64.whl (9.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.2/9.2 MB 16.9 MB/s eta 0:00:00
Requirement already satisfied: pillow in c:\users\nico\appdata\local\programs\python\python310\lib\site-packages (from wxPython->robotframework-ride) (8.4.0)
Requirement already satisfied: numpy in c:\users\nico\appdata\local\programs\python\python310\lib\site-packages (from wxPython->robotframework-ride) (1.21.4)
Requirement already satisfied: six in c:\users\nico\appdata\local\programs\python\python310\lib\site-packages (from wxPython->robotframework-ride) (1.16.0)
Installing collected packages: Pywin32, wxPython, PyPubSub, Pygments, psutil, robotframework-ride
  DEPRECATION: robotframework-ride is being installed using the legacy 'setup.py install' method, because it does not have a 'pyproject.toml' and the 'wheel' package is not installed. pip 23.1 will enforce this behaviour change. A possible replacement is to enable the '--use-pep517' option. Discussion can be found at https://github.com/pypa/pip/issues/8559
  Running setup.py install for robotframework-ride ... done
Successfully installed PyPubSub-4.0.3 Pygments-2.15.1 Pywin32-306 psutil-5.9.5 robotframework-ride-2.0.5 wxPython-4.2.0

[notice] A new release of pip is available: 23.0.1 -> 23.1.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

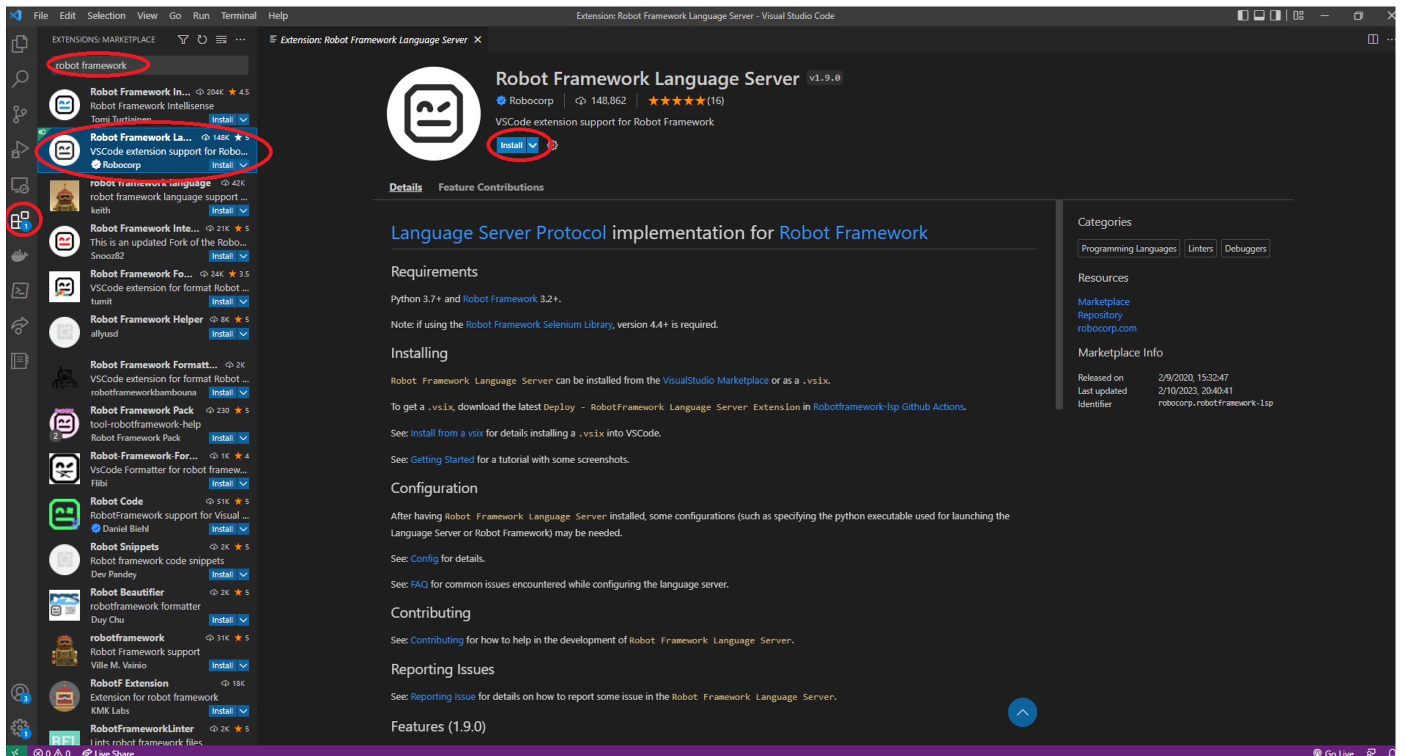
Kun asennus on suoritettu loppuun, RIDEn voi käynnistää komennolla ”ride.py”.



RIDEn voi lisätä myös työpöydälle ohjelman ylävalikosta



Visual Studio Codeen asennetaan lisäosa, joka mahdollistaa testien suorituksen suoraan ohjelmasta käsin. Lisäksi se tunnistaa Robot Frameworkin syntaksin, jolloin ohjelmakoodin kehitys helpottuu huomattavasti. Lisäosa löytyy Visual Studio Coden Extensions-välilehdeltä ohjelman vasemmasta reunasta. Ylhäällä olevaan pieneen hakukenttään kirjoitetaan ”robot framework” ja valitaan Robocorpin kehittämä ”Robot Framework Language Server”. Tämän jälkeen lisäosan sivu avautuu ja asennus suoritetaan klikkaamalla sinisestä ”Install”-painikkeesta



Liite 3: Testeissä käytetyt SeleniumLibrary-avainsanat ja niiden vastineet Browser librarystä

Selenium	Browser
Add Cookie	Add Cookie
Capture Page Screenshot	Take Screenshot
Checkbox Should Be Selected	Get Checkbox State ${selector} == checked
Choose File	Upload File By Selector
Clear Element Text	Clear Text
Click Button	Click
Click Element	Click
Click Link	Click
Close All Browsers	Close Browser
Create Webdriver	*Ei vastinetta*
Delete All Cookies	Eat Cookies
Element Should Be Visible	Get Element States ${selector} contains visible
Element Should Contain	Get Text ${selector} contains ${value}
Element Should Not Be Visible	Get Element States ${selector} contains hidden
Element Text Should Be	Get Text ${selector} == value
Execute Javascript	Evaluate Javascript
Get Browser Ids	Get Browser Ids
Get Cookie	Get Cookie
Get Element Attribute	Get Attribute
Get Element Count	Get Count
Get List Items	Get Select Options
Get Location	Get URL
Get Source	Get Page Source
Get Text	Get Text
Get Title	Get Title
Get Value	Get Text
Get Vertical Position	Get BoundingBox
Get WebElement	Get Element
Get WebElements	Get Elements
Go to	Go To
Input Text	Fill Text/Type Text
Location Should Contain	Get URL contains value
Maximize Browser Window	Ei suoraan vastaavaa avainsanaa. Set Viewport Size muuttaa ruudun kokoa, mutta fullscreen määritetään browserin käynnistymisvaiheessa
Mouse Over	Hover
Open Browser	New Browser
Page Should Contain	Get Text ${selector} contains ${value}
Page Should Contain Button	Get Element States ${selector} contains attached
Page Should Contain Element	Get Element States ${selector} contains attached
Page Should Contain Link	Get Element States ${selector} contains attached
Page Should Not Contain	Get Element States ${selector} contains detached

Page Should Not Contain Button	Get Element States ${selector} contains detached
Page Should Not Contain Element	Get Element States ${selector} contains detached
Page Should Not Contain Link	Get Element States ${selector} contains detached
Press Keys	Press Keys/Keyboard Key
Register Keyword To Run On Failure	Register Keyword To Run On Failure
Reload Page	Reload
Scroll Element Into View	Scroll To Element
Set Focus To Element	Focus
Set Window Size	Set Viewport Size
Switch Browser	Switch Browser/Switch Context/Switch Page
Wait For Condition	Wait For Condition
Wait Until Element Contains	Wait For Condition Text ${selector} contains ${value}
Wait Until Element Is Enabled	Wait For Elements State ${selector} enabled
Wait Until Element Is Not Visible	Wait For Elements State ${selector} hidden
Wait Until Element Is Visible	Wait For Elements State ${selector} visible
Wait Until Location Contains	Wait For Condition Url contains ${value}
Wait Until Page Contains	Wait For Elements State ${selector} attached
Wait Until Page Contains Element	Wait For Elements State ${selector} attached
Wait Until Page Does Not Contain	Wait For Elements State ${selector} detached
Wait Until Page Does Not Contain Element	Wait For Elements State ${selector} detached