OULUN AMMATTIKORKEAKOULU

Tuomas Riihiaho

WEB-AUTOMATISOINNIN VERTAILU: SELENIUM, PLAYWRIGHT JA BROWSE RLIBRARY

WEB-AUTOMATISOINNIN VERTAILU: SELENIUM, PLAYWRIGHT JA BROWSE RLIBRARY

Tuomas Riihiaho
Opinnäytetyö
Kevät 2024
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, Ohjelmistokehitys

Tekijä: Tuomas Riihiaho
Opinnäytetyön nimi: Web-automatisoinnin vertailu: Selenium, Playwright ja BrowserLibrary
Työn ohjaaja(t): Jukka Nevalainen & Jaakko Lankila
Työn valmistumislukukausi ja -vuosi: Kevät 2024 Sivumäärä: 54

Opinnäytetyössä tarkasteltiin testiautomaatiokirjastojen käyttökelpoisuutta web-sovellusten testauksessa. Näitä testiautomaatiokirjastoja olivat SeleniumLibrary, BrowserLibrary ja Playwright. SeleniumLibrary ja BrowserLibrary toimivat Robot Framework -kehyksessä, kun taas Playwright otettiin käyttöön samassa projektissa ilman kyseistä kehystä itsenäisenä kirjastona Python-ohjelmointikielellä. Tavoitteena oli toteuttaa suunniteltu testitapaus jokaisella testiautomaatiokirjastolla erikseen ja vertailla niiden suorituskykyä, tuloksia sekä käytettävyyttä. Tutkimalla näiden kolmen kokonaisuutta pyrittiin selvittämään, mikä olisi paras tapa toteuttaa testiautomaatiot. Testitapauksen valinta perustui puhtaasti käyttöliittymän hallintaan, eikä siihen liittynyt vaativampaa testitapauksen mukauttamista toimimaan rajapintoja vasten.

Testiautomaatiokirjastoja tutkiessa tutustuttiin, miten ne teoriassa toimivat ja miten ne otettiin käyttöön kehitysympäristöissä. Tämän jälkeen suunniteltiin testiautomaatiokirjastojen yhteiseen käyttöön testitapaus Jirassa. Testitapaus suunniteltiin yrityksen uuden Reskontra-sovelluksen käyttöliittymän osiosta.

Lopuksi arvioitiin valmiiden testitapausten tuloksia testiautomaatiokirjastojen välillä keskenään. Loppupohdinnassa havaittiin, että testiautomaatioiden toteuttaminen modernimmilla kirjastoilla säästää aikaa ja resursseja sekä nopeuttaa testitapausten suorittamista. Päädyttiin ratkaisuun, että on erityisen suotavaa käyttää päivitetympiä testiautomaatiokirjastoja testiautomaatioiden rakentamisessa ja että olemassa olevat testitapaukset voidaan päivittää vaivattomasti käyttämään uudempaa testiautomaatiokirjastoa pienellä työmäärällä.

Asiasanat: Testiautomaatio, Robot Framework, SeleniumLibrary, Playwright, BrowserLibrary

ABSTRACT

Oulu University of Applied Sciences
Bachelor's degree in information technology

Author: Tuomas Riihiaho
Title of thesis: Web-automation comparison: Selenium, Playwright ja BrowserLibrary
Supervisor(s): Jukka Nevalainen & Jaakko Lankila
Term and year when the thesis was submitted: Spring 2024
Number of pages: 54

The thesis explored the suitability of different test automation libraries for testing web applications. These test automation libraries included SeleniumLibrary, BrowserLibrary and Playwright. SeleniumLibrary and BrowserLibrary run on the Robot Framework, while Playwright was implemented in the same project without that framework as a standalone library in the Python programming language. The goal was to implement the test case with each test automation library separately and compare their performance, results, and usability. By examining the entirety of these three, I tried to find out what would be the best way to implement test automation. The selection of the test case was based purely on user interface management and did not involve the more demanding adaptation of the test case to work against APIs.

When studying test automation libraries, I got to know how they work in theory and how they were implemented in development environments. After this, a test case was planned for the joint use of test automation libraries in Jira. The test case was designed from the user interface section of the company's new Ledger application.

Finally, the results of the finished test cases were evaluated between the test automation libraries. In the final discussion, it was found that implementing test automation with more modern libraries saves time and resources and speeds up the execution of test cases. I concluded that it is particularly desirable to use more updated test automation libraries when building test automations and that existing test cases can be easily updated to use a newer test automation library with a small amount of work.

Keywords: Test automation, Robot Framework, SeleniumLibrary, Playwright, BrowserLibrary

SISÄLLYS

1 JOHDANTO

Nykypäivänä testaus on keskeinen osa sovelluskehitystä, joka varmistaa sovellusten laadun ja suorituskyvyn ennen niiden julkaisemista. Testauksen tarkoituksena on tunnistaa ja korjata virheitä hyvissä ajoin jo sovellusten kehitysvaiheessa sekä vähentää riskejä ja varmistaa sovelluksen toimivuus erilaisissa käyttöolosuhteissa. Testausta voidaan suorittaa joko manuaalisesti henkilön suorittamana tai automatisoidusti käyttäen ohjelmistoskriptejä.

Opinnäytetyössäni tutkitaan web-pohjaisten sovellusten käyttöliittymien automaatiotestausta ja vertaillaan tuloksia sekä suoriutumista eri testityökalujen välillä. Vaikka pääpaino onkin yrityksen valitsemalla testiympäristöllä, joka on Robot Framework, tutkin myös Playwrightin ominaisuuksia ja vertaan niitä Robot Frameworkin tarjoamien testikirjastojen suorituskykyyn sekä tuloksiin. Tämän vertailun avulla pyritään arvioimaan kunkin työkalun soveltuvuutta erilaisiin testausvaatimuksiin ja ympäristöihin sekä tunnistamaan vahvuudet ja heikkoudet työkalujen käytössä.

Työn toimeksiantajana on suomalainen tilitoimistopalveluita tarjoava yritys, joka tarjoaa myös talousohjelmistoja ja -palveluita yrityksille. Ennen opinnäytetyön aloittamista suoritin harjoittelujakson yrityksen myyntireskontran kehitystiimissä automaatiotestaajana. Projektitiimeillä on ollut käytössä testiautomaatiokehityksessä avoimen lähdekoodin rajapinta Robot Framework, jossa tiimit ovat käyttäneet SeleniumLibraryä.

Ajan myötä minulle selvisi, että kirjasto on ollut useiden vuosien ajan yleisin valinta testiautomaatiokehityksessä. Perehdyttyäni ajan kanssa SeleniumLibraryyn huomasin kuitenkin, että markkinoille oli tullut uusia modernisoituja ja optimoidumpia tapoja toteuttaa automaatiotestit. Päätin siten valita testiautomaatiokirjastojen vertailun web-automatisoinnissa opinnäytetyöaiheekseni.

Tarkastelun kohteena on yrityksen reskontraprojektiin suunniteltu testitapaus. Projekti on jo ennestään itselleni tuttu, koska työskentelin testiautomaatiotehtävissä projektin parissa harjoittelujakson aikana. Kokemukset ja havainnot, joita olen kerännyt testitapauksen integroimisesta eri testiautomaatiokirjastoihin, muodostavat opinnäytetyön loppupohdinnan perustan.

2 OHJELMISTOTESTAUS

Ohjelmistotestauksen tarkoituksena on osoittaa, että ohjelma tekee sen, mitä sen on tarkoitus tehdä, ja havaita ohjelmavirheet, ennen kuin se otetaan käyttöön tuotannossa. Kun ohjelmistoa testataan, suoritetaan testit ohjelmaa vasten keinotekoisella datalla. Testauksessa tarkkaillaan tulosten virheitä, poikkeavuuksia tai tietoja ohjelman ei-toiminnallisista ominaisuuksista. (1, s. 227.)

2.1 Ohjelmistotestauksen tavat ja virheiden löytäminen

Ensimmäisenä tavoitteena on osoittaa kehittäjälle ja asiakkaalle, että ohjelmisto täyttää sille asetetut vaatimukset. Mukautettuja ohjelmistoja varten tämä merkitsee vähintään yhden testin tekemistä jokaiselle vaaditulle toiminnolle. Tavallisille ohjelmistotuotteille se tarkoittaa, että kaikki järjestelmän ominaisuudet, jotka ovat sisällytetty tuotejulkaisuun, on testattava. Ominaisuuksien yhdistelmiä voidaan myös testata, millä tarkastellaan ei-toivottua vuorovaikutusta niiden välillä. (1, s. 227.)

Toiseksi pyritään löytämään syötteitä tai syötesarjoja, joissa ohjelmiston käyttäytyminen on virheellinen, ei-toivottu tai ei vastaa sen määrityksiä. Nämä johtuvat ohjelmiston virheistä tai virheellisestä toteutuksesta, mikä voi pahimmillaan viedä ominaisuuden takaisin suunnittelupöydälle. Testatessa ohjelmistoa vikojen löytämiseksi pyritään kitkemään ei-toivottua järjestelmän toimintaa, kuten järjestelmän kaatumisia, ei-toivottuja vuorovaikutuksia muiden järjestelmien kanssa, virheellisiä laskelmia ja tietojen korruptiota. (1, s. 227.)

Näistä edellä mainituista tavoista ensimmäinen on nimeltään validointitestaus, missä odotetaan järjestelmän toimivan oikein käyttämällä testitapauksia, jotka kuvaavat sen odotettua käyttöä. Toista metodia kutsutaan vikojen testaukseksi, jossa testitapaukset on suunniteltu paljastamaan viat. Vikatestauksen testitapaukset voivat olla tarkoituksellisesti epämääräisiä, eikä niiden tarvitse välttämättä edustaa järjestelmän normaalia käyttöä. Näiden tapojen välillä ei ole kuitenkaan selvää rajaa, vaan on otettava huomioon, että validointitestauksen aikana voidaan huomata järjestelmässä vikoja ja vikatestauksen aikana osa testeistä osoittaa, että ohjelma täyttää sille asetetut vaatimukset. (1, s. 227.)

2.2 Testisuunnitelma

Suunnitelman tarkoitus on osoittaa, miten käyttöönotettavan ohjelmiston laatu saadaan varmistettua. Testaussuunnitelma tehdään samaan aikaan projektisuunnitelman kanssa alkuvaiheessa. (2.) Testauksen dokumentointi on avainasemassa suunnitteluprosessissa. Testit on tärkeä dokumentoida huolellisesti, jotta resurssienhallinta pysyy hallinnassa. Testisuunnitelmalla on ratkaiseva merkitys onnistuneen testauksen lopputuloksen varmistamiseksi. (3.)

2.2.1 Testisuunnitelman merkitys projektin laadun varmistuksessa

Suunnitteluprosessi alkaa testisuunnitelmalla, joka määrittelee suunnitelman ja strategian sovelluksen testaamiseen. Suunnitelman tarkoitus on varmistaa laadun, luotettavuuden ja toimivuuden ennen sen julkaisemista tai käyttöönottoa. Projektitasolla testaussuunnitelmassa määritellään testattavat kohteet, niiden testausmenetelmät ja testaustyypit, jotka jaetaan testaajien kesken. Uusien projektien alkaessa yrityksissä tiimien testipäälliköt laativat testisuunnitelman. Varhaisessa vaiheessa on hyvä tehdä alustava suunnitelman osuus, joka määrittelee tekijät ja roolit prosessissa (kuva 1). Valmis suunnitelma jaetaan kaikille projektiin osallistuville. (3.)

Tekijät	Roolit
Kuka kirjoittaa testisuunnitelmat?	Testijohto, testipäällikkö, testausinsinööri
Kuka arvioi testisuunnitelman?	Testijohtaja, testipäällikkö, testausinsinööri, asiakas, kehitystiimi
Kuka hyväksyy testisuunnitelman?	Asiakas, testipäällikkö
Kuka kirjoittaa testitapauksia?	Testijohto, testausinsinööri
Kuka arvioi testitapauksia?	Testausinsinööri, testijohtaja, asiakas, kehitystiimi
Kuka hyväksyy testitapaukset?	Testauspäällikkö, testijohto, asiakas

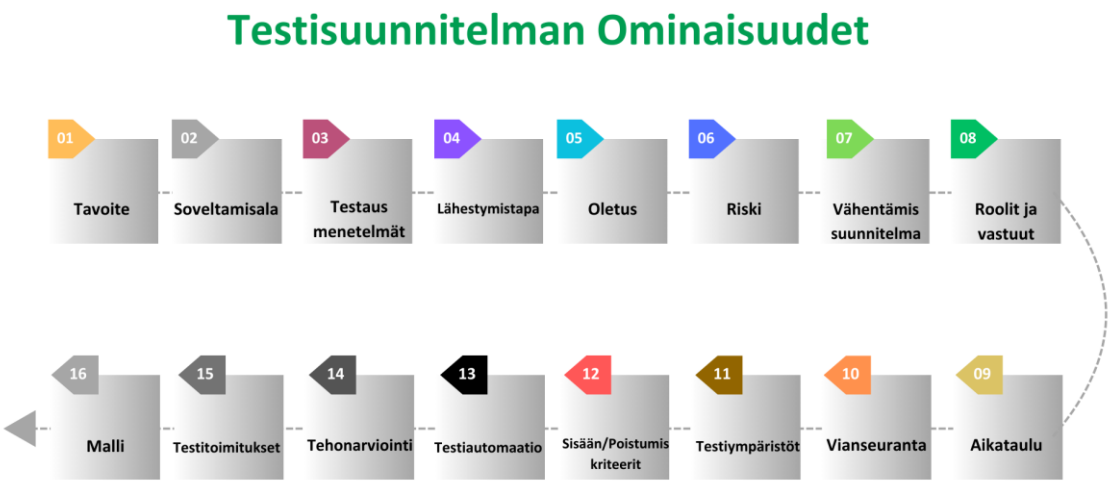
KUVA 1. Testisuunnitelma: Tekijät ja Roolit testauksessa (mukaillen 3)

2.2.2 Testisuunnitelman hyödyt ja tavoitteet

Testisuunnitelma on olennainen osa testausprosessia. Laadunvarmistusinsinööreille suunnitelma on selkeä ohje testaustoimintojen suorittamiseen. Sen avulla rajataan ylimääräisiä testitoimintoja,

arvioidaan aikaa, kustannuksia ja vaivaa testauksessa. Suunnitelma auttaa myös aikatauluttamaan testaustoiminnot. Lisäksi se on uudelleenkäytettävä dokumentti, joka sisältää tärkeitä näkökohtia, kuten testin laajuuden ja strategian, ja sitä voidaan käyttää uudelleen muissakin projekteissa. (3.)

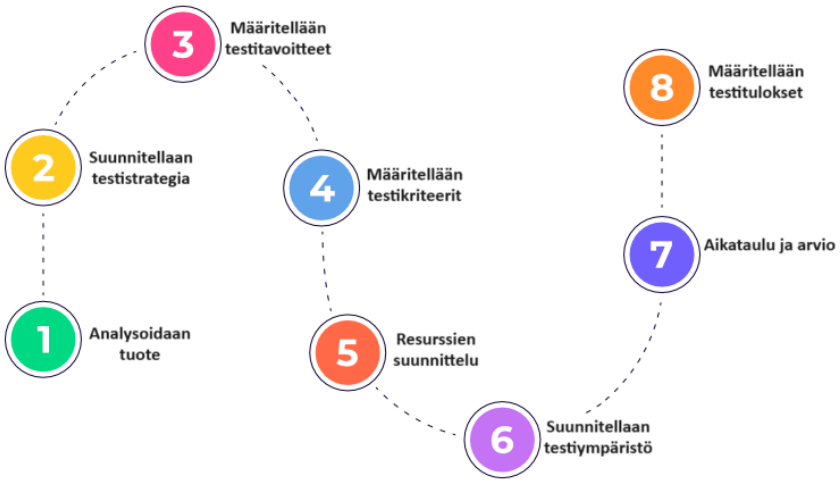
Testisuunnitelman tavoitteena on saada laaja yleiskatsaus testaustoiminnoista ja luoda aikataulu sekä arvion tarvittavista resursseista. Suunnitelmaa seurataan testauksissa projektin alusta loppuun asti. Se auttaa löytämään ratkaisuja ja toimii sääntökirjana projektin vaiheittaiselle edistymiselle. Suunnitelman laatimiseen ei ole kuitenkaan kiveen hakattua sääntöä, millainen se pitäisi olla, mutta 16 testisuunnitelman ominaisuutta voivat toimia avuksi sen rakentamisessa. (Kuva 2) (3.)



KUVA 2. Testaussuunnitelma: Osat ja ominaisuudet (mukaillen 3)

2.2.3 Testisuunnitelman luominen

Testisuunnitelman luomisessa voidaan noudattaa kahdeksaa eri vaihetta (kuva 3) (3).



KUVA 3. Testisuunnitelman luonti (mukaillen 3)

Ensimmäisessä vaiheessa analysoidaan tuote, keskittyen tuotteen analysointiin, asiakkaiden, suunnittelijoiden ja kehittäjien haastatteluun sekä tuotteen esittelyyn. Tässä vaiheessa vastataan kysymyksiin, kuten mikä on tuotteen ensisijainen tavoite, kuka käyttää tuotetta, mitkä ovat tuotteen laitteisto- ja ohjelmistotiedot sekä miten tuote toimii. (3.)

Toisessa vaiheessa suunnitellaan testausstrategia, jossa testistrategia-asiakirja laaditaan ja siinä käsitellään testauksen laajuutta, testaustyyppiä, riskejä ja ongelmia sekä testilogistiikkaa. Laajuudessa rajataan, mitä komponentteja testataan, ja mitkä testaukset voidaan ohittaa. Testaustyyppi sisältää erilaiset testityypit, jotka sisällytetään projektin testauksiin. Dokumentoidaan riskit ja ongelmat, joissa kirjataan ylös kaikki mahdolliset riskit, joita voi esiintyä testauksissa. Lisäksi dokumentoidaan myös testilogistiikka, joka sisältää testaajien nimet ja heidän suorittamat testit. (3.)

Kolmannessa vaiheessa määritellään testin tavoitteet, mikä on kriittinen vaihe testausstrategian suunnittelussa. Tässä vaiheessa määritellään selkeästi testien suorittamisen tavoitteet ja odotetut lopputulokset. Tavoitteisiin sisältyy kattava luettelo ohjelmiston ominaisuuksista, kuten toiminnallisuudesta, graafisesta käyttöliittymästä, suorituskyvystä ja turvallisuudesta. (3.)

Neljännessä vaiheessa määritellään testikriteerit. Testikriteereitä on kaksi pääkategoriaa: keskeytyskriteerit ja poistumiskriteerit. Keskeytyskriteereissä määritellään vertailuarvot kaikkien testien

käyttäytymisille. Poistumiskriteereissä määritellään testivaiheen vertailuarvoja tai onnistuneen projektin loppuunsaattamisessa osoittamat vertailuarvot. Molemmat ovat odotettuja tuloksia, ja tulosten on vastattava toinen toistaan ennen kuin siirrytään seuraavaan vaiheeseen. (3.)

Viidennessä vaiheessa on resurssien suunnittelu. Suunnittelussa luodaan yksityiskohtainen luettelo kaikista tarvittavista resursseista projektin loppuunsaattamiseksi. (3.)

Kuudennessa vaiheessa suunnitellaan testiympäristö, joka varmistaa oikeiden laitteiden ja ohjelmistojen saatavuuden testauksen suorittamiseksi todellisissa käyttöolosuhteissa. Tärkeä vaihe, koska testausympäristöjen täytyy olla oikeita laitteita, joihin on asennettu web-selaimia käyttöjärjestelmiin, jotta testaajat voivat tarkkailla ohjelmistojen käyttäytymistä realistisissa käyttöolosuhteissa. (3.)

Seitsemännessä vaiheessa on aikataulu ja arvio. Tässä vaiheessa jaetaan projekti pienempiin tehtäviin ja varataan aikaa ja vaivaa kullekin tehtävälle tehokasta ajan arviointia varten. Yritykset käyttävät usein projektinhallintatyökaluohjelmistoja tässä vaiheessa. (3.)

Kahdeksannessa vaiheessa määritellään testitulokset. Tulokset viittaavat luetteloon asiakirjoista, työkaluista ja laitteista, jotka on luotava, toimitettava ja ylläpidettävä testaustoimien tukemiseksi. (3.)

3 TESTIAUTOMAATION HYÖDYT JA HAASTEET

Testiautomaatiota voidaan käyttää automatisoimaan toistuvia testauksia kehitettävässä ohjelmistossa. Se on arvokas työkalu ohjelmistokehittäjille, jotka työskentelevät saman ohjelmiston parissa testiautomaatioinsinöörien kanssa, ja se voi huomattavasti helpottaa sovelluksen kehityskulkua löytämällä virheet jo varhaisessa kehitysvaiheessa esimerkiksi regressiotestausten avulla. (2.) Tulosten avulla voidaan kehittää ohjelmiston laatua ja sen perusteella testausautomaatiota voidaankin pitää eräänlaisena laadunvarmistustoimenpiteenä (6).

3.1 Testiautomaation hyödyt

Ohjelmistojen kehitysprojekteissa on usein huomattava määrä testattavaa niiden laajuuden vuoksi ja määrä vain kasvaa projektin edetessä. Tästä syystä kannattaa automatisoida testejä niin paljon kuin mahdollista. Tilanteissa, joissa virheitä ei havaita ajoissa, ne ehtivät pidempään vaikuttaa ohjelman muiden osien toimintaan, ja korjaaminen on työläämpää ja kalliimpaa. Tehokkain tapa ehkäistä virheiden syntyä on havaita mahdollisia virhetilanteita jo suunnittelu- ja määrittelyvaiheessa. Testiautomaation toistuvilla testeillä säästetään myös huomattavia aikamääriä verrattuna siihen, jos pitäisi testata aina manuaalisesti. (7.)

3.2 Omat kokemukset testiautomaation haasteista

Haasteena on usein testien ylläpito. Ohjelmiston muuttuessa myös testit on usein päivitettävä vastaamaan uusia vaatimuksia ja toiminnallisuuksia. Automaatiotestit ovat herkkiä pienimmillekin muutoksille ohjelmiston koodissa ja saattavat sitä myötä hajota, mikä sitten vaatii palaamista jo tehdyn testin pariin ja korjata se. Sovelluksen kehittäjät voivat ottaa huomioon testiautomaation tarpeet lisäämällä testitunnisteita sivuston elementeille, jotta automaatiotestien rakentaminen olisi sujuvampaa.

4 TESTIAUTOMAATION OSA-ALUEITA

Tässä luvussa käsitellään testiautomaation osa-alueita ja niihin liittyviä tärkeimpiä asioita. Ensimmäisenä käydään läpi manuaalitestaus, joka on tärkeä ymmärtää käsitteenä ennen muita osa-alueita.

4.1 Manuaalitestaus

Manuaalisessa testauksessa varmistetaan, että suoritettu ohjelmisto toimii kuten on suunniteltu, suorittamalla ohjelmiston prosesseja tai toimintoja. Manuaalitestaukset suoritetaan käyttöliittymässä asiakkaan tavoin kokeilemalla kehitysvaiheessa olevaa tuotetta. Samalla tietäen, mitä pitäisi tapahtua ja miten ohjelmiston kuuluisi käyttäytyä. (2.)

Testit usein dokumentoidaan projektinhallintatyökaluun testitapaus-pohjiksi, joiden dokumentaatiota päivitetään ohjelmiston kehityksen edetessä. Testitapaukset ovat ennalta suunniteltuja ja jaoteltuja toimenpiteiden mukaisesti. Prosessi pilkotaan testitapauksen askeleisiin, joka on usein yksi suoritettava toiminto. Esimerkki testitapaus voisi olla seuraavanlainen: Lisätään X myyntitilaukselle tuotetta Y 15kpl. Testitapaukset määritellään yleensä siten, että ne suorittavat yhden toiminnon kerrallaan. (2.)

Testitapauksiin kirjataan myös odotettu lopputulos, jolla varmistetaan, että järjestelmä toimii niin kuin sen pitää. Aiemmasta esimerkistä kirjattaisiin testitapaukseen: Varmista, että myyntitilaukselle X tuli Y tuotetta 15kpl. Suunnitellut testitapaukset ovat hyödyllisiä testauksen laadun, luotettavuuden ja kattavuuden arvioinnin takia. Jos testitapaukset eivät ole suunniteltuja, ne ovat epäluotettavia ja virheitä voi päästä ohjelmiston julkaisuun asti. (2.) Testitapauksia on hyvä manuaali testata ennen niiden automatisointia ohjelmistoskripteillä eli testiautomaatiolla. Huomioitavaa on myös, että kaikkia testitapauksia ei välttämättä voida automatisoida.

4.2 Regressiotestaus

Toisin kuin manuaalitestaus ja automaatiotestaus, regressiotestaus ei ole erillinen testauksen muoto vaan se on yleistermi, mikä tarkoittaa uudelleentestaamista. Regressiotestauksesta puhutaan, kun toimivan järjestelmän osia muutetaan kehityksen vaiheessa. Valmiit testitapaukset toimivat edelleen oikein muutosten jälkeen. Regressiotestauksen käsitettä voidaan soveltaa myös tilanteissa, joissa kehitettävä järjestelmä saavuttaa osatavoitteensa, ja halutaan varmentaa kaikkien toimintojen oikeellisuus huomioon ottaen myös ne, jotka olivat jo valmiita aikaisemmassa versiossa. Tärkeimpänä ominaisuutena on kuitenkin todentaa, että jo ennestään korjatut ongelmat eivät ole enää läsnä komponenttiin tehtyjen muutosten myötä, sekä että mitään muuta uutta ei ole mennyt rikki. (15, s. 68–69.)

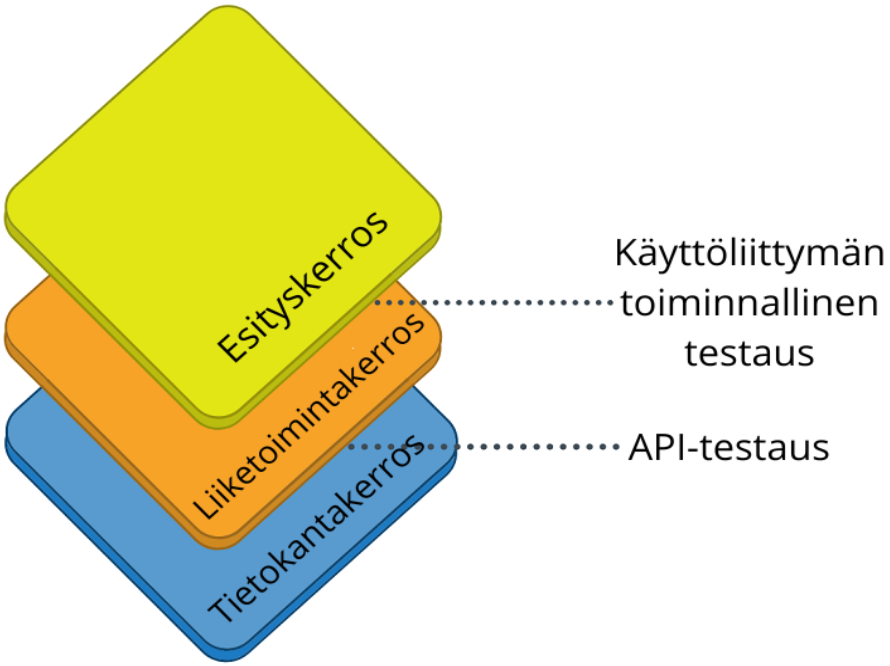
4.3 API-testaus

”API on lyhenne sanoista application programming interface, eli ohjelmointirajapinta ja tarkoittaa samaa kuin rajapinta” (16).

Rajapinnat mahdollistavat ohjelmistojen välisen tiedonsiirron eli integraation. Rajapintojen avulla voidaan lähettää pyyntöjä ohjelmistoille, josta halutaan hakea tai johon halutaan viedä tietoja. Tietoturvallisista syistä rajapinnat suojataan API-avaimilla, jotta kuka tahansa ei pääsisi käsiksi käyttämään niitä. Usein rajapintojen välillä saatetaan välittää sensitiivistä tietoa, jotka pidetään yksityisenä. (16.)

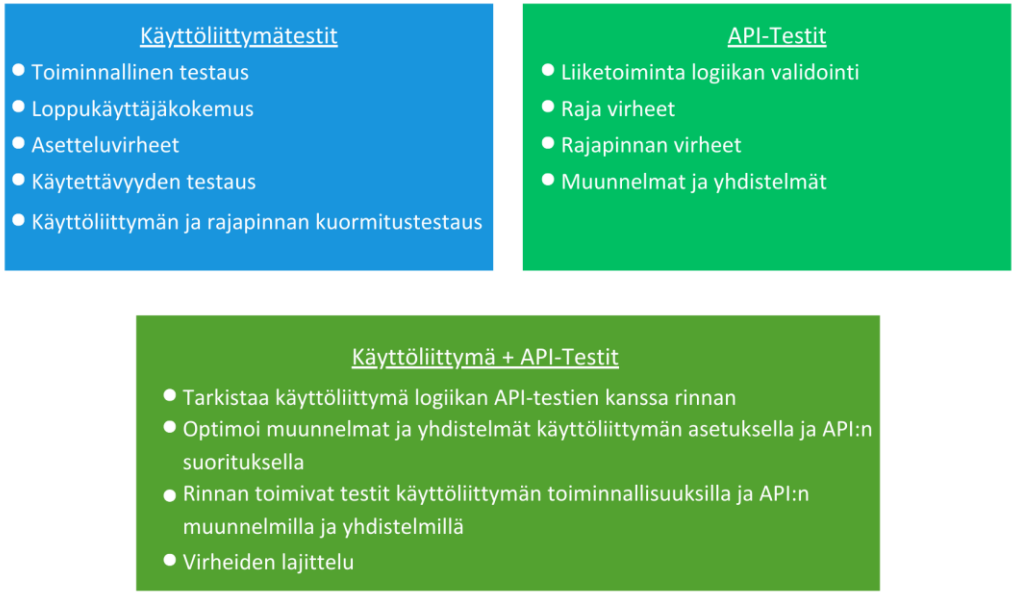
API-testaus on prosessi, mikä varmistaa, että rajapinta toimii halutulla tavalla. Ohjelmistokehittäjät voivat testata rajapintoja manuaalisesti tai ne voidaan myös automatisoida ohjelmistoskripteillä. API-testausta voidaan tehdä missä tahansa kehitysvaiheen aikana ja ne ovatkin tärkeitä testattavia ohjelmiston laadun kannalta. Virheen tapahtuessa rajapintakerroksessa ohjelmistossa voi johtaa käyttäjäkohtaisiin virheisiin tai tiedon viiveeseen heikentäen asiakkaiden luottamusta. Kyseiset ongelmat asettavat kehittäjille painetta tarjota sovelluksia, jotka ovat aina saatavilla ja toimivat nopeasti ongelmitta. (17.)

Web-ohjelmistot koostuvat kolmikerroksisesta sovellus arkkitehtuurista. Ylä-osassa on esityskerros, joka sisältää käyttöliittymän. Käyttöliittymä näyttää sisällön ja tiedot loppukäyttäjälle. Keskimmäisenä on liiketoimintakerros, joka sisältää toiminnallisen liiketoimintalogiikan. Alimmaisena on tietokantakerros, joka koostuu tietojen tallennusjärjestelmästä. (Kuva 4) (18.)



KUVA 4. Kolmikerroksinen sovellus arkkitehtuuri (mukaillen 18)

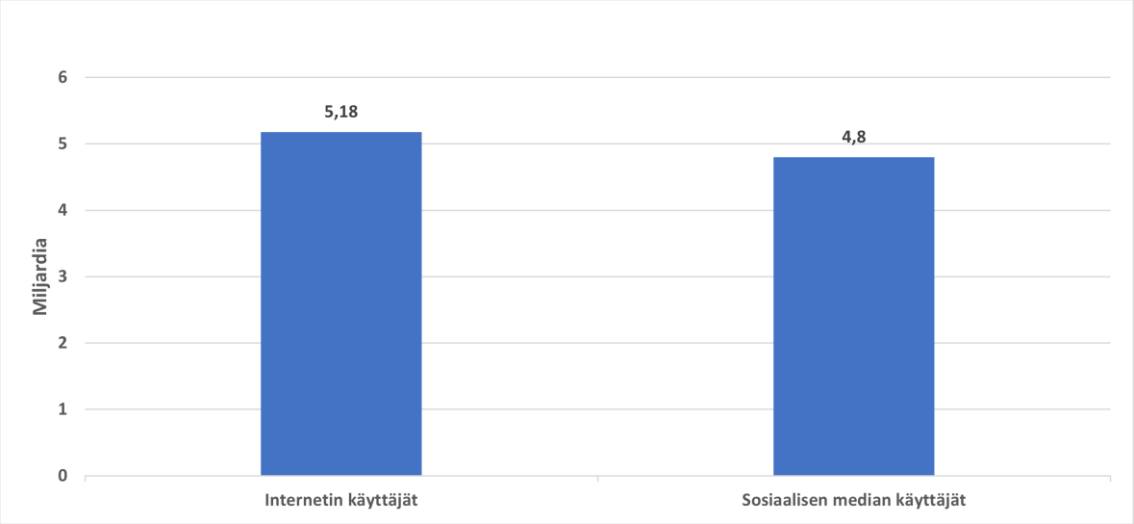
Automatisoidessa käyttöliittymää rajapinnat työskentelevät huomaamatta taustalla ohjelmiston toimintojen tarpeen mukaan. Automatisoinnissa saattaa joskus olla tarpeen käyttää rajapintoja rinnakkain testien kanssa, jolloin mukautettu rajapintapyyntö rakennetaan ohjelmistoskriptiin ennen varsinaista käyttöliittymätestin suoritusta. Testitapaukset, jotka näitä vaiheita vaativat, saadaan toteutettua halutulla tavalla ja voidaan varmentaa, että liiketoimintalogiikka toimii odotetusti. (18.)



KUVA 5. Käyttöliittymä- ja API-testien hyödyt yhdessä ja erikseen (mukaillen 18)

5 WEB-SOVELLUKSET TESTAUSKOHTEENA

Internet, World Wide Web (WWW) tai yksinkertaisesti Web on maailmanlaajuinen toisiinsa kytkettyjen tietoverkkojen järjestelmä, joka on maailman vallankumouksellisin tietotekniikan ja sovellusten kehitysalusta. Internet on ollut maailmanlaajuisessa suosiossa jo kymmeniä vuosia ja yhä enemmän tekemisissä ihmisten päivittäisessä elämässä. Statistiikkojen mukaan internetin käytön määrä on globaalisti kasvamaan päin vuosittain. Huhtikuussa 2023 Statistan tehdyssä tutkimuksessa, esitellään lukuja käyttäjämääristä (kuva 6). (8.)



KUVA 6. Graafi Internetin ja sosiaalisen median käyttäjistä (mukaillen 8)

5.1 Web-sovelluksien hierarkia

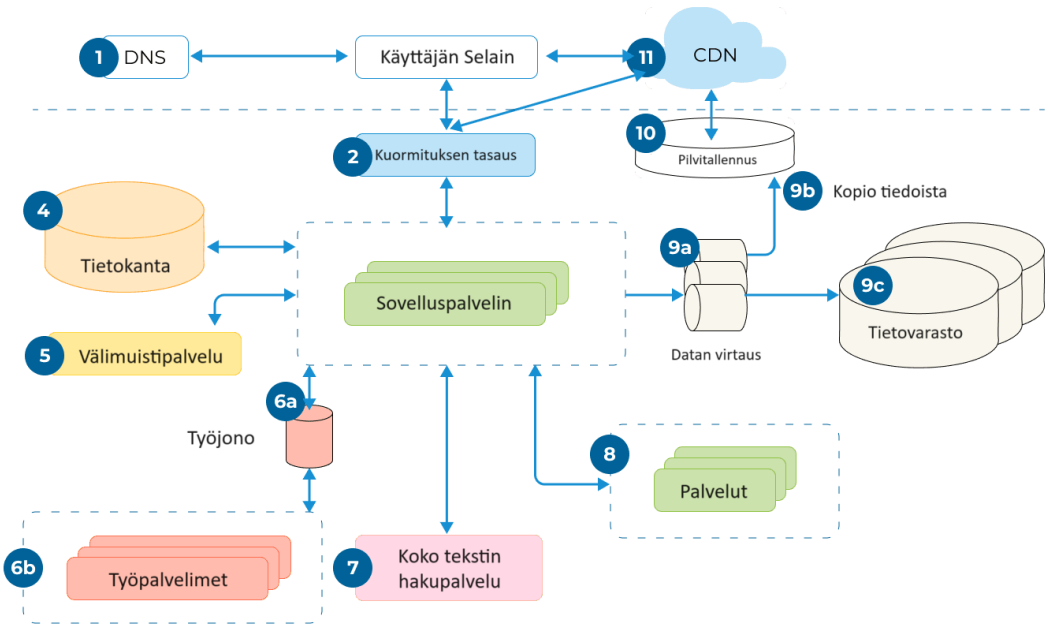
Internetissä verkkosovellukset tai verkkosivustot ovat asiakaspuolelle ladattuja kokonaisuuksia, jotka sisältävät useita kerroksia toisistaan riippuvaisia moduuleja. Sivustot on rakennettu ydinverkkoteknologioiden perustalle. (8.)

- HTML: Hypertext Markup Language tai HTML on tunnistepohjainen avoimesti standardoitu asiakirjamerkintäjärjestelmä, joka määrittää selaimen dokumenttiobjektimallin (DOM) kootun rakenteen ja sen komponentit. (8.)

- CSS: CSS on vakiintunut tapa määritellä DOMin ulkoasua ja asetteluja verkossa. Se käyttää syntakseja, jotka kuvaavat, miten eri HTML-elementit näytetään selaimessa. Näitä syntakseja voidaan suoraan kohdistaa HTML-elementteihin, jotta sivuston ulkoasua voidaan muotoilla kehittäjän tarpeiden mukaan. (8.)
- JavaScript: JavaScript on ohjelmointikeli, jolla toteutetaan mutkikkaita ominaisuuksia verkkosivustoilla. Aina kun verkkosivusto tekee jotain missä tapahtuu staattisen sijaan jotain dynaamista, voidaan olla varmoja, että JavaScript on käytössä. Vakioverkkoteknologioiden kolmas osuus HTML:n ja CSS:n rinnalla. (9.)

Edeltäviin ydinteknologioihin on sisällytetty useita teknologioita toimintakyvyn lisäämiseksi. Suosituista JavaScript-kehyksistä ja kirjastoista, kuten käyttöliittymäkerrokset React, Angular, ja Vue, sekä muissa verkkoon liittyvissä teknologioissa, alueet ovat todella laajoja. Käyttöliittymäkerroksista käytetään nimitystä Frontend. (8.)

Käyttöliittymäkerroksien lisäksi sovelluksissa on tyypillisesti tausta- tai palvelinpuolen kerros, joista puhutaan nimityksellä Backend. Backend määrittelee, kuinka sovelluksen eri osiot kommunikoivat keskenään pyyntöjen avulla, jotka palauttavat tarvittaessa vastauksia. Sitä käytetään dataliikenteen hallintaan, mitä tietoja näytetään esimerkiksi käyttöliittymässä. Kuvassa 7 on moderni sovellushierarkia hahmoteltuna. (14.)



KUVA 7. Moderni verkkosovellusarkkitehtuurikaavio (mukaillen 10)

5.2 Web-sovellusten haavoittuvuus

Yhä useammat yritykset on nykypäivänä ovat siirtyneet käyttämään verkkopohjaisia sovelluksia. Verkkosovelluksiin liittyy kuitenkin riskejä verrattuna perinteisiin työpöytäsovelluksiin. (11.) Verkkosovellukset altistuvat hyökkäyksien kohteeksi joka vuosi. Hyökkääjät kehittävät uusia tapoja vaarantamaan arkaluonteisia tietoja ja päästääkseen murtamaan kohteiden tietokantaa. Tietoturva-asiantuntijoilla on paljon tehtävää saada pidettyä verkkosovellukset suojattuina korjaamalla löydettyjä haavoittuvuuksia ja vahvistamalla järjestelmien suojauksia joka vuosi. (12.)

5.3 Web-sovellusten testaus

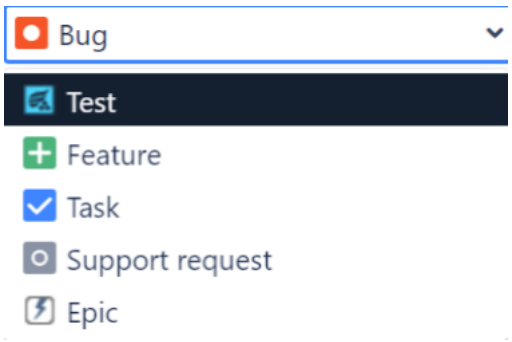
Ohjelmistosuunnittelussa testaustyypit ja tekniikat voidaan suorittaa sovelluksen vaatimusten mukaan. Verkkosivuston toiminnallisuustestaus on käytäntö, joka sisältää testauskohteita sovelluksen eri osioista. Käyttöliittymät, rajapinnat, tietokannat, tietoturva, asiakas- ja palvelintestaukset kuuluvat sivuston perustoimintojen kanssa toiminnallisuustestaukseen. Toiminnallinen testaus tarjoaa erinomaisen tavan varmistaa verkkosivustojen kaikkien ominaisuuksien toimivuus, ja sen avulla yrityksen testaajat voivat suorittaa sekä manuaalista että automaattisia testauksia näille kohteille. Toiminnallisuustestauksen vaiheet ovat monipuolisia työkaluja laaduntarkastuksessa ja itsessään jo kattava testaustapa sovellukselle. (13.)

6 PROJEKTINHALLINTATYÖKALUNA JIRA

Projektinhallintatyökalu, kuten Jira, on olennainen osa projektien toteutusta sen jälkeen, kun testaukset on suunniteltu ja valmisteltu. Jira on australialaisen ohjelmistoyrityksen Atlassianin kehittämä ohjelmistosovellus (4), jonka avulla tiimit voivat seurata ja hallita projektin työtehtäviään, suunnitella ja aikatauluttaa tehtäviään, seurata edistymistä ja raportoida tuloksista. Jira on erinomainen tehtävienhallintaohjelmisto-sovellus projekteille, sillä se kykenee havainnollistamaan tarpeelliset vaiheet sovellusten kehityksessä. Sen käyttö on helppoa ja yksinkertaista, mikä tekee siitä miellyttävän työkalun projektien hallintaan. Kaikki alkaa projektin luomisesta, jonka jälkeen projektin eri osiot ja tehtävät voidaan luoda projektin sisälle. (5.)

6.1 Työkohteet Jirassa

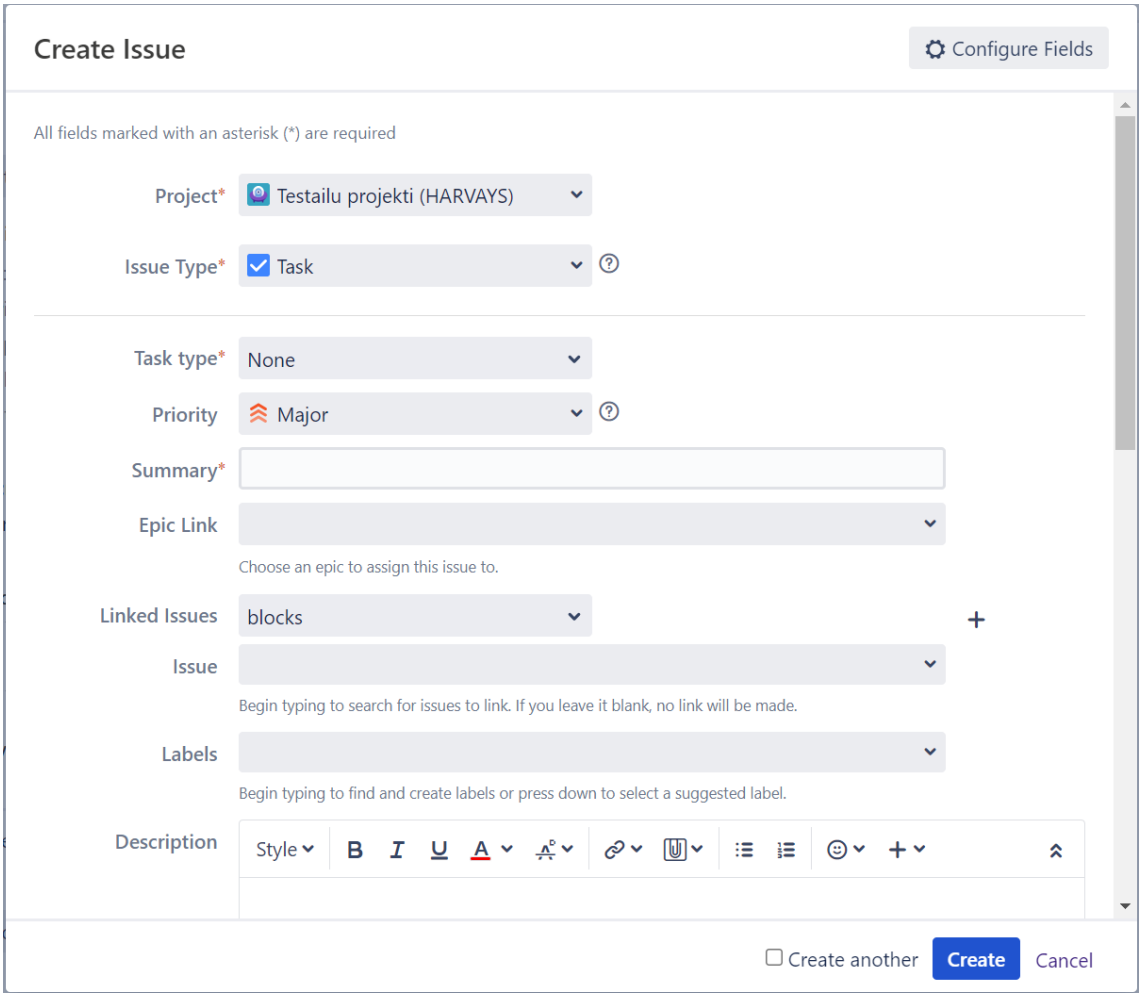
Jirassa työtehtävä (Issue) on yksittäinen työkohde, jota seurataan luomisesta valmistumiseen (4). Sille voidaan merkitä tiiminjäsen vastuuhenkilöksi, jolloin nähdään, kuka työskentelee sen parissa. Työskentelemässäni projektissa yleisimmät työtehtävät ilmoitetaan Jirassa muodoissa Epic, Ominaisuus (Feature), Bugi (Bug), Testi (Test) tai Tehtävä (Task) (kuva 8). On myös olemassa muitakin työtehtävien ilmoitusmuotoja.



KUVA 8. Työtehtävän tyypit Jirassa

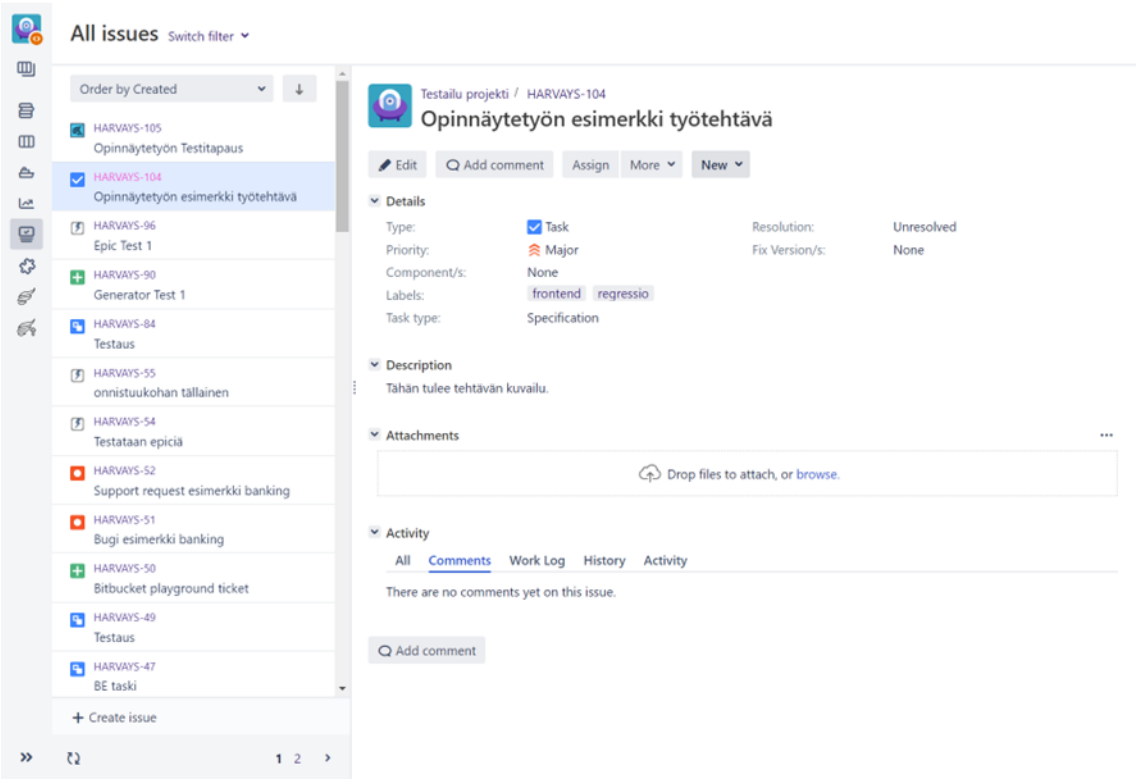
6.2 Testien dokumentointi Jirassa

Testitapauksen dokumentointi tapahtuu Jirassa luomalla uusi työtehtävä valitussa projektissa. Luonti-ikkunassa valitaan, minkä projektin alle työtehtävä laitetaan. Työtehtävätyypin valinta tapahtuu myös tässä vaiheessa, samoin kuin tärkeysjärjestys ja testin kuvaus. Kuvassa 9 on käytetty humoristista esimerkkiprojektia nimeltään Härväys.



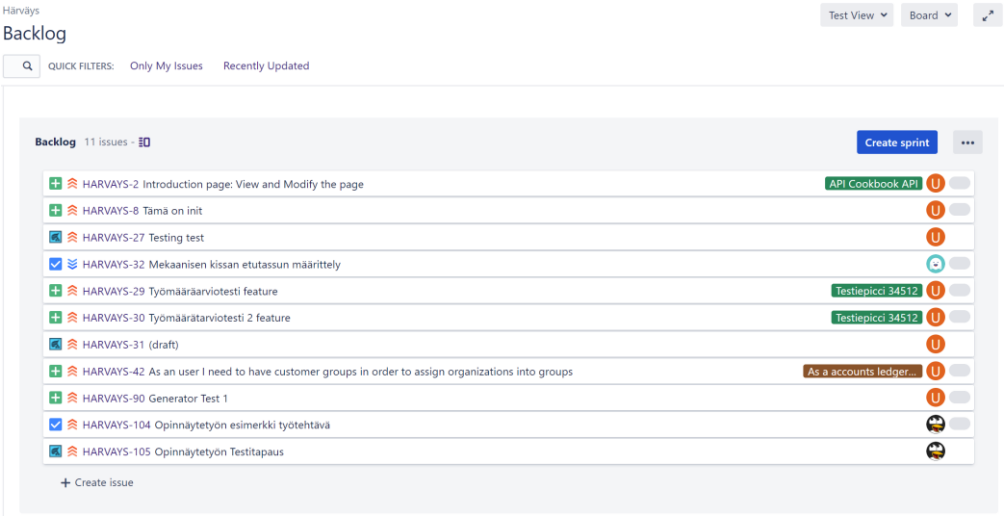
KUVA 9. Työtehtävän luonti ikkuna Jirassa

Tehtävän luonnin jälkeen nähdään työtehtävälistalta äskettäin luotu työtehtävä. Tehtävässä nähdään tietoja työtehtävästä, tehtävän luojan nimi, sen kuvailu ja mahdolliset työvaiheet. Lisäksi voidaan merkitä työtehtävälle tekijä kohdassa Assignee.



KUVA 10. Jiran työtehtävälista ja luotu työtehtävä

Projektin työtehtäviä voidaan tarkastella projektin backlogista, jotta projektissa toimivat työntekijät pysyvät ajan tasalla siitä, mitkä työtehtävät ovat tällä hetkellä käynnissä (kuva 11). Härväys-projektissa ei ole määritelty sprinttejä, mutta työtehtävät lisätään yleensä tavanomaisesti sprinteille projektinhallinnassa. Sprintti on enintään kuukauden kestävä työjakso ketterässä kehitysmenetelmässä. Ohjelmistokehitysprojekteissa tiimit käyttävät usein projektinhallintamenetelmiä, joissa sprinttejä seurataan.

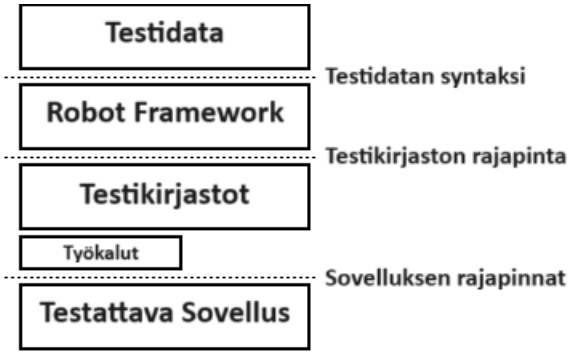


KUVA 11. Projektin backlog Jirassa

7 ROBOT FRAMEWORK JA LIITÄNNÄISKIRJASTOT

Robot Framework on avoimen lähdekoodin testiautomaatiokehys, joka on helposti lähestyttävä, helppokäyttöinen ja tarjoaa avainsanalähtöistä automaatiota ohjelmistoille. Kehys perustuu Python-ohjelmointikieleen ja se mahdollistaa testaajalle rakentaa korkeamman tason testitapauksia, jotka sujuvasti voidaan kääntää koneellisesti suoritettaviksi automaatio skripteiksi. Kehyksessä on laajasti erilaisia työkaluja ja kirjastoja, joiden avulla kehys sopii erinomaisesti monen tyyppisiin testauksiin. Robot Framework soveltuu kätevästi uudelleenkäytettävän koodin hallintaan, minkä ansiosta sekä teknisesti osaavien, että vähemmän osaavien henkilöiden on helppo kirjoittaa ja ylläpitää testejä yhteistyössä kehitystiimin kanssa. (19.) Robot Framework julkaistiin avoimena lähdekoodina yleisön käyttöön vuonna 2008. (20).

Toiminnallisuus Robot Frameworkissa perustuu erilaisiin testauskirjastoihin ja niiden käyttö mahdollistaa monenlaisia käyttömahdollisuuksia automaatiotesteille. Testaajan tarpeisiin voi kuulua esimerkiksi mobiilisovellusten automatisointi tai vuorovaikutus tietokannan kanssa. Kehyksestä löytyykin näihin asioihin tarkoitetut kirjastot. Vaikka osa kirjastoista on sisäänrakennettu kehyksen ytimeen, testaajat voivat myös luoda omia kustomoituja kirjastoja. Kehyksen arkkitehtuuri (kuva 12) havainnollistaa yleisen suorituksen toimintaperiaatetta. (19.)



KUVA 12. Robot Frameworkin arkkitehtuuri (mukaillen 20)

7.1 SeleniumLibrary

SeleniumLibrary on yksi suosituimmista Robot Frameworkin testauskirjastoista web-sovellusten automatisointiin. SeleniumLibrary hyödyntää WebDriver-moduuleja hallinnoimaan selaimen käyt-

täytymistä. (21.) WebDrivereita tarvitaan etenkin, kun halutaan suorittaa verkkosovellusten testaamista eri selaimilla. Testaaja voi vapaasti valita Seleniumin tukemista selaimista: Google Chrome, Mozilla Firefox, Safari, Internet Explorer ja Microsoft Edge. (22.)

SeleniumLibraryssä on huomattava määrä sisäänrakennettuja avainsanoja, joiden käyttö perustuu verkkosivuilla olevien elementtien paikantamiseen. Kun Robot Frameworkissä kirjaston skriptiä kirjoitetaan, avainsanat käyttävät lähes poikkeuksetta nimettyä argumenttia elementin paikantamiseen, mitä halutaan käsitellä. Tätä argumenttia kutsutaan paikantimeksi (Locator). Paikantimet kertovat avainsanalle, mitä elementtiä halutaan käyttää toimenpiteessä. On monia eri tapoja kirjoittaa paikannin avainsanalle ja se valitaan usein sen perusteella, miten sivusto onkaan rakennettu. Selaimen kehittäjäkonsolia käyttäen on kätevä selvittää, millä tavalla paikannin määritellään avainsanalle. Tärkeää on myös kirjoittaa paikannin mahdollisimman tarkasti kirjoitussääntöjen mukaan, jotta vältytään erilaisilta virhetilanteilta. (21.)

7.2 BrowserLibrary

Robot Frameworkin BrowserLibrary on kehyksen oma verkkosovellusten testaamiseen suunniteltu kirjasto kuten SeleniumLibrary, mutta toteutettu uudella modernilla lähestymistavalla. Kirjasto on kehitetty Playwright-nimisen automaatiokehyksen pohjalta ja sen avulla hallinta toteutetaan ohjelmointirajapintaa hyödyntäen. Sen tarkoitus on päivittää web-testaus 2020-luvulle. (23.)

BrowserLibrary toimii avainsana pohjaisesti aivan kuten SeleniumLibrarykin, mutta huomattavilla eroilla. Seleniumissa avainsanat tekevät ainoastaan yhtä asiaa ja pyrkii tekemään sen mahdollisimman hyvin. BrowserLibraryssä avainsanat ovat monitoiminnallisia ja voivat tehdä yhdellä avainsanalla monta eri asiaa, jolloin testin kirjoitusrakenne lyhenee ja optimoituu. (24.)

Paikantimina BrowserLibraryssä on vakiovalintana css-tunnisteet, mutta myös toisia paikantimia voi käyttää, kuten SeleniumLibraryssäkin. BrowserLibraryn kehittäjät kertovat, että odotuksia sisältävät avainsanat, joita käytetään laajasti SeleniumLibraryssä, voidaan jättää pois, sillä kirjasto hoitaa odottelut automaattisesti. (24.)

Tutkin myöhemmässä luvussa, vastaavatko testien suorituskykyyn liittyvät väitteet todellisuutta, ja millaista on käyttää kirjastoa käytännössä.

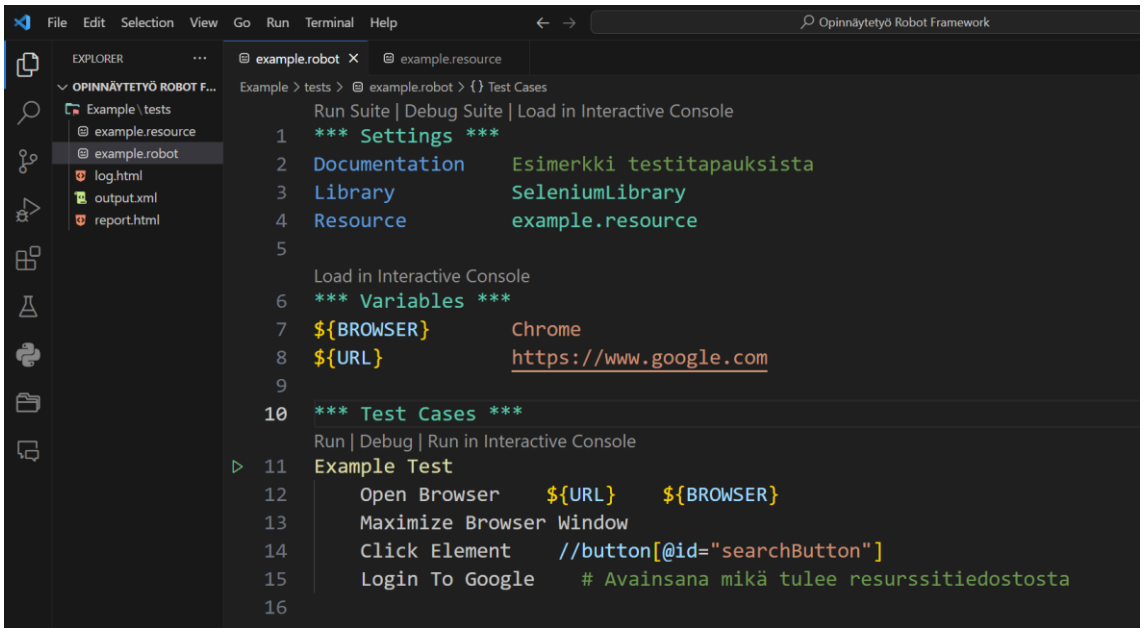
7.3 Testidatan rakentaminen ja kirjoitussäännöt Robot Frameworkissa

Ohjelmointiympäristössä toimitaan seuraavanlaisesti, kun Robot Framework, valittu testiautomaatiokirjasto ja muut tarvittavat paketit on asennettu. Luodaan testitapaukset robot-päätteisiin tiedostoihin. Luotu robot-päätteinen tiedosto muodostaa testisarjan (Test Suite). Testisarjaan luodaan yleensä useampi aiheeseen liittyvä testitapaus (Test Cases), jotka suorittavat yhden määritellyn testitapauksen kerrallaan. Samaan testisarjaan on myös hyvä luoda resource-päätteinen tiedosto. Resurssitiedostoon luodaan kaikki monimutkaisempi ohjelmallinen toimintalogiikka ja robot-päätteinen tiedosto toimiikin pelkästään logiikkojen järjestelijänä ja testitapauksen suorituksen ajajana. Ohjelmointiympäristössä on tärkeää järjestellä testisarjat järkevästi luettavuuden ja tapausten organisoinnin takia. (20.)

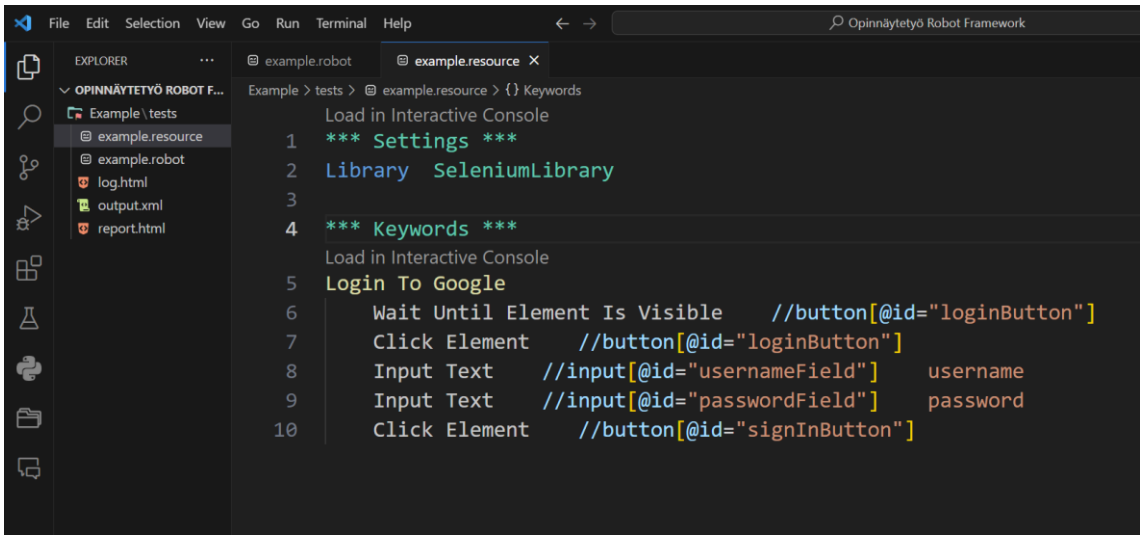
Kirjoitussäännöt menevät selkeästi Robot Frameworkissä. Yleisin kirjoitusmuoto on väleillä eroteltu formaatti, jossa skriptin palaset, kuten avainsanat ja niiden argumentit ovat eroteltu toisistaan useammalla välilyönnillä tai tabuloinnilla. Testitiedostoissa on myös määritellyt osioalueet, jotka kuvataan otsikoilla. (Kuva 13) (20.)

Osion otsikko	Käyttötarkoitus
*** Settings ***	• Testikirjastojen, Resurssitiedostojen ja muuttuja tiedostojen tuonti • Metatietojen määrittäminen testisarjoille ja testitapauksille
*** Variables ***	Muuttujien määrittely, joita voidaan käyttää muissa osioissa
*** Test Cases ***	Testitapausten määrittely käytettävissä olevista avainsanoista
*** Tasks ***	Samankaltainen käyttötarkoitus kuin Test-Cases-osiossa. Tiedosto voi sisältää vain toisen näistä.
*** Keywords ***	Tiedoston omien avainsanojen osio
*** Comments ***	Osio kommenteille, joita Robot Framework ei huomioi

KUVA 13. Testitiedoston eri osiot (mukaillen 20)



KUVA 14. Esimerkki robot-päätteisestä tiedostosta VS Codessa



KUVA 15. Esimerkki resource-päätteisestä tiedostosta VS Codessa

Kuvassa 14 nähdään esimerkki robot-päätteisestä tiedostosta. Kirjoitusmuotona on käytetty väleillä eroteltua formaattia, kuten nähdään tabuloinnista avainsanan jälkeen. Huomataan, että kaikki avainsanat testitapauksessa tulevat SeleniumLibraryn sisältä paitsi Login To Google -avainsana, joka on määritelty resurssitiedostossa (kuva 15). Testitapaus ei varsinaisesti kuvaa oikeata kirjautumista Googlen sivulla, mutta antaa hyvän esimerkin, miltä testitapaus voisi esimerkiksi näyttää.

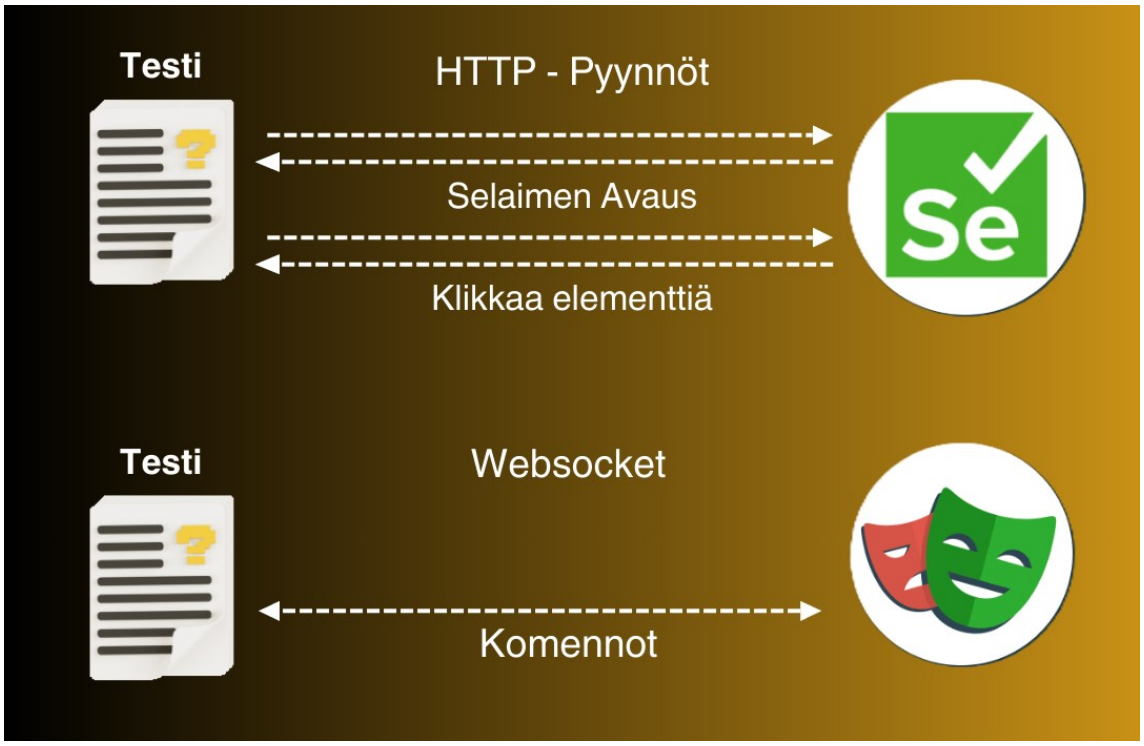
Testitapausten suoritusten jälkeen Robot Framework luo testitiedot yksinkertaiseen taulukkomuotoon sen luomaan raporttiin (20).

8 PLAYWRIGHT

Microsoftin luoma Playwright on avoimen lähdekoodin kehys, joka kykenee automatisoimaan Chromium-, Firefox- ja WebKit-pohjaisia selaimia yhden rajapinnan avulla. Vuonna 2020 julkaistu kehys saavutti nopeasti testaajien suosion markkinoilla. Playwright sisältää moderneita uusia ominaisuuksia, mikä tekee siitä vahvan kilpailijan toisille kehyksille. Testaajat, joilla on halu vaihtaa Seleniumista Playwrightiin, pystyvät tekemään sen vaivattomasti, sillä Playwrightissa on tuki useille ohjelmointikielille, joita voidaan käyttää myös Seleniumissa. (25.) Playwrightin moderneissa työkaluissa on muun muassa Codegen, Playwright Inspector ja Trace Viewer. (26).

Kehyksen mukana tulee Codegen-työkalu, jolla voidaan luoda automaattisesti testisyntaksia nauhoittamalla selainta käyttäjän toimesta ja kopioida generoidun nauhoituksen syntaksi sen testitapauksen ajoa varten. Codegen avaa kaksi ikkunaa, selainikkunan, jossa käyttäjä on vuorovaikutuksessa verkkoselaimen kanssa sekä Playwright Inspector -ikkunan, jossa nähdään reaaliajassa generoitu koodisyntaksi selaimen käytön nauhoituksen mukaan. Koodisyntaksin voi kääntää halutuun ohjelmointikieleen, jota Playwright tukee. (27.)

Trace Viewer on graafinen työkalu. Työkalun avulla voidaan tutkia suoritetun testin jälkeen tarkemmin, mitä testin eri vaiheissa tapahtui. Testin vaiheita tarkastaessa voidaan siirtyä eteenpäin tai taaksepäin jokaisen toiminnon kohdalla. (28.)



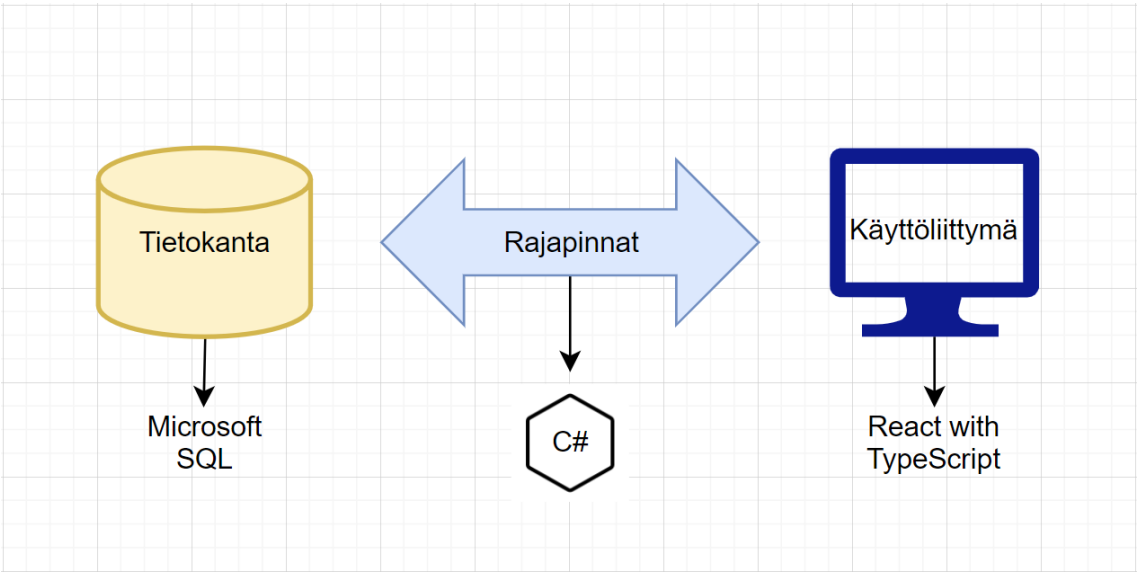
KUVA 16. Playwrightin arkkitehtuuri (mukaillen 25)

Playwrightin arkkitehtuuria ymmärretään, kun sitä verrataan Seleniumin arkkitehtuuriin. Seleniumin arkkitehtuuri lähettää komennot itsenäisinä http-pyyntöinä ja vastaanottaa JSON-tyyppisiä vastauksia. Jokaista komentoa kohden tehdään siis oma http-pyyntö. Tämän seurauksena testit hidastuvat ja komentoja joudutaan odottamaan pidempään. Playwrightin tapauksessa avataan yksi Websocket-yhteys ja kaikki pyynnöt kulkevat sen kautta, mikä mahdollistaa testien nopeamman suorituksen. (25.)

Tutkin myöhemmässä luvussa, millaista on käyttää Playwrightia ja kuinka paljon testien suoriutuminen kehittyy.

9 KÄYTTÖLIITTYMÄTESTIN VALMISTELU

Opinnäytetyöni käytännön osuudessa vertailen aiemmissa luvuissa kerrottuja web-automatisoinnin kirjastoja käyttöliittymätestissä suorituskyvyltään ja käytännön tuntumalta, millaista niitä on käyttää ja mitä eroavaisuuksia kirjastoilla on. Käyttöliittymätestin aluksi suunnitellaan projektinhallintatyökalu Jiraan testitapaus, josta kirjoitetaan automaatiotesti jokaisella kirjastolla erikseen. Lisäksi kerron, mitkä kaikki ohjelmistot pitää asentaa, jotta voidaan ajaa testiautomaatioita ohjelmointiympäristössä virheettä. Ohjelmointiympäristönä toimii jokaisen kirjaston testitapauksen toteutuksessa Visual Studio Code. Reskontra-sovelluksen arkkitehtuuri, mihin testitapaus kohdistetaan, on hahmoteltu kuvassa 17. Sovelluksen rakennuksessa on käytetty erilaisia teknologioita, kuten Tietokannan hallintaan Microsoft SQL Serveriä, rajapintojen ohjelmointiin C# -ohjelmointikieltä ja käyttöliittymän rakennukseen React -kirjastoa, joka käyttää TypeScript -ohjelmointikieltä.



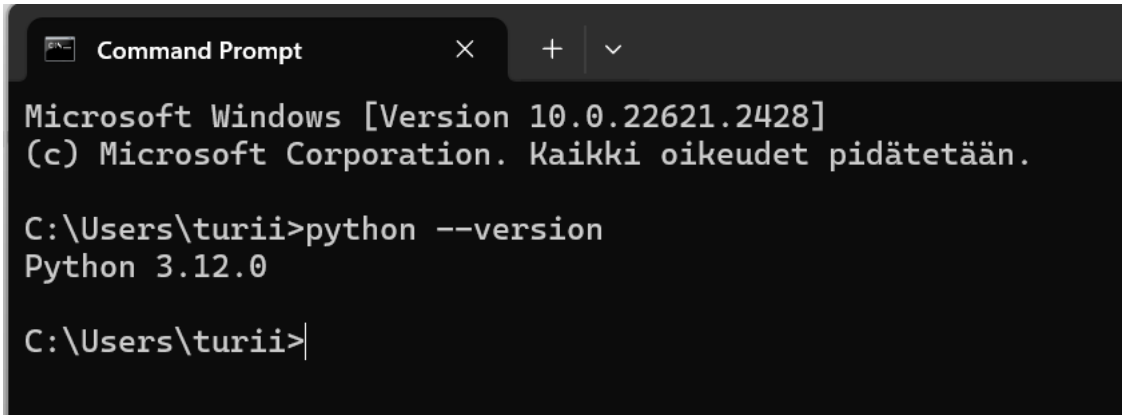
KUVA 17. Reskontra-sovelluksen arkkitehtuuri

9.1 Python ja Robot Framework

Robot Framework pohjautuu Python-ohjelmointikieleen ja sen on oltava asennettuna tietokoneelle Robot Frameworkin asennuksen yhteydessä. Python on syytä lisätä myös tietokoneen ympäristömuuttujiin niille tietokoneille, joissa on Windows-käyttöjärjestelmä, jotta voidaan käyttää Pythonia

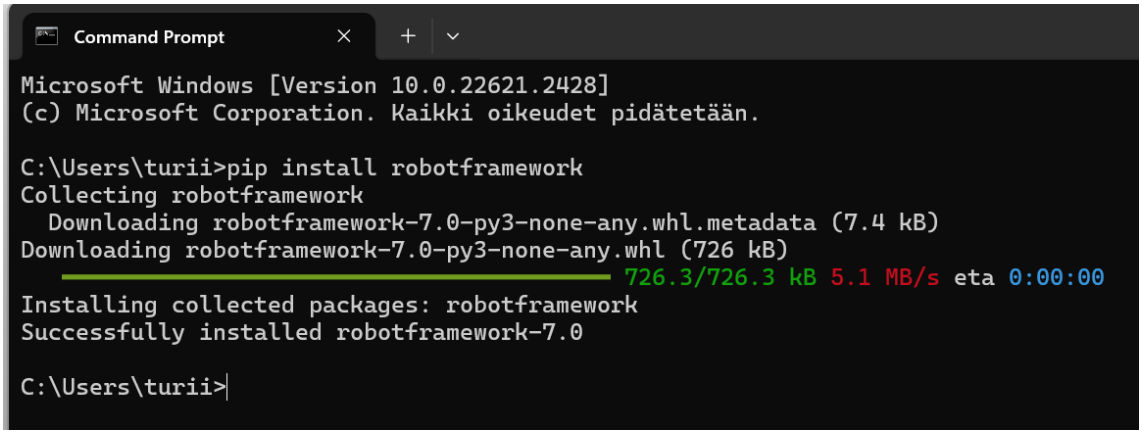
suoraan komentoriviltä. Lisäystä ympäristömuuttujiin kysytään käyttäjältä, kun Pythonia asennetaan. Pythonin voi asentaa tietokoneelle sen omilta kotisivuilta https://www.python.org/downloads/. (20.)

Kun Python on asennettu, voidaan sen asennuksen onnistuminen varmentaa komentoriviltä syöttämällä komento **python –version** (kuva 18).



KUVA 18. Pythonin asennuksen varmistus komentoriviltä

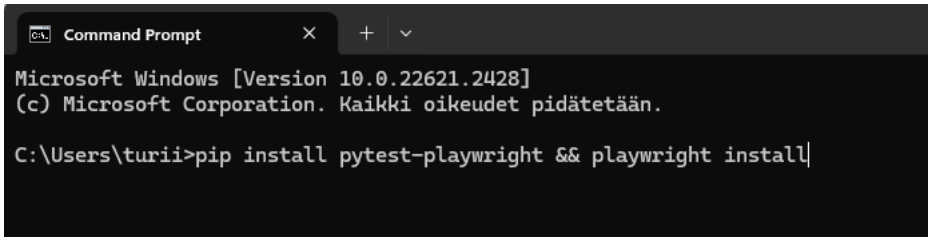
Pythonin asennuksen mukana tulee pip-pakettien hallinta- ja asennustyökalu, jolla voidaan asentaa tai poistaa lisättyjä Python-moduuleita ja kirjastoja. Tätä asennustyökalua käyttämällä voi komentoriviltä asentaa Robot Framework -kehyksen komennolla **pip install robotframework**. (Kuva 19) (20.)



KUVA 19. Robot Frameworkin asennus

9.2 Playwright ja Node.js

Playwright suosittelema tapa testien kirjoittamiseen on käyttää virallista Playwright Pytest -laajennusta, joka mahdollistaa kontekstin eristämisen ja hyödyntää sitä eri selainkokoonpanoissa. Playwrightin asennuksen aloittaminen tapahtuu komentoriviltä syöttämällä komento **pip install pytest-playwright**. Asennukseen tarvitaan myös tarvittavat selaimet komennolla **playwright install**. Molemmat komennot voidaan laittaa yhdelle riville komentorivillä käyttäen && loogista operaattoria (Kuva 20) (29.)



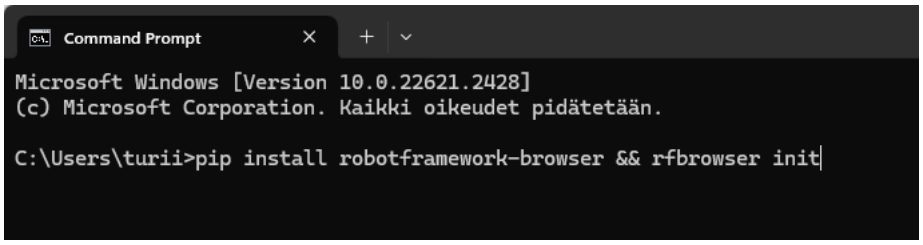
KUVA 20. Playwrightin ja selainlaajennusten asennus

Playwright vaatii Node.js -ajoympäristön asennuksen toimiakseen. Node.js:n voi asentaa osoitteesta https://node.js.org/en.

9.3 BrowserLibrary

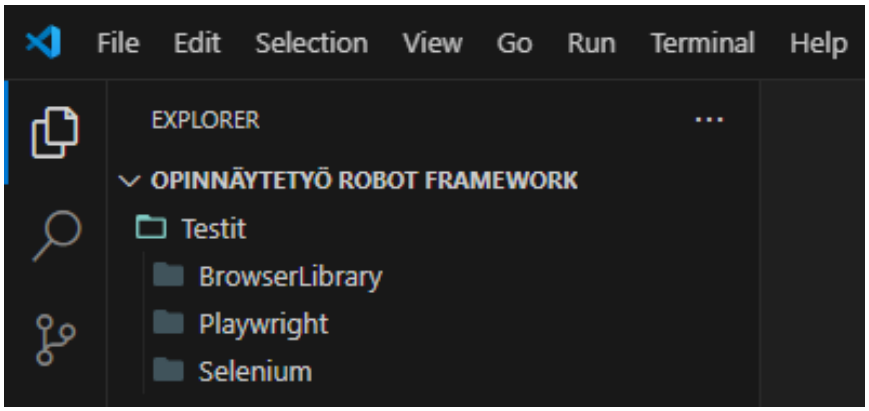
BrowserLibraryn esivaatimuksena on myös Node.js:n asennus, koska se on Playwrightin pohjalle rakennettu kirjasto. Kun tämä on tehty, voidaan BrowserLibraryn asennus aloittaa. Asennus voidaan suorittaa kahdella eri tavalla. Ensimmäisessä tavassa asennus tapahtuu selaimien binääritiedostojen kanssa, ja toisessa ohitetaan selain asennuskomennosta. Parempi vaihtoehto on asentaa selainten binääritiedostojen kanssa, jos päätavoitteena on kehittää testitapauksia. (30.)

BrowserLibraryn asennus selainten binääritiedostojen kanssa tapahtuu syöttämällä komentoriville komento **pip install robotframework-browser**. Tämän lisäksi alustetaan selaimen kirjasto komennolla **rfbrowser init**. Molemmat komennot voidaan ajaa käyttäen samaa loogista operaattoria kuin aiemmin. (Kuva 21) (30.)



KUVA 21. BrowserLibraryn ja selainkirjaston asennus

Luodaan vielä Visual Studio Codeen projekti, johon tehdään asianmukaiset kansiot hierarkiaan kirjastojen mukaisesti (kuva 22).



KUVA 22. Kansiohierarkia opinnäytetyön käytännön osuudesta

Eri kirjastojen asentelut on nyt tehty, joilla voidaan kirjoittaa ja ajaa automaatiotestejä kolmella eri tavalla. Seuraavassa luvussa suunnitellaan testitapaus Jirassa, johon kirjoitetaan testitapauksen eri vaiheet ja odotetut lopputulokset. Käytän testitapauksen kohteena yrityksen reskontraprojektia, jonka parissa työskentelin jo harjoittelujakson aikana.

10 SUUNNITTELU JIRASSA

Suunnittelin testitapauksen seuraavaksi Jiraan ja päädyin valitsemaan testitapauksen, jossa puhtaasti keskityttiin käyttöliittymän hallintaan, eikä siihen liittynyt vaativampaa testitapauksen mukauttamista toimimaan rajapintoja vasten. Tarkastelin yrityksen reskontra-sovellusta, jossa on Microsoftin Azure AD-kirjautuminen ennen web-sivulle pääsyä, millä on useampia eri osioita sovelluksen eri toiminnallisuuksien mukaan. Reskontra-sovelluksen Saatavat -sivun osiossa on taulukko, johon ladattiin laajasti organisaatioiden saatavia järjestyksessä taulukon eri riveille. Taulukolla on monia erilaisia toimintoja tietojen katselmointiin. Taulukon toiminnallisuuksiin kuului muun muassa yhden tai useamman saatavan valinta, tietojen laajentaminen ja tiivistäminen valinnoille, saatavan tai saatavien lisätietojen katselmointi taulukon rivin laajentamisen jälkeen alitaulukossa. Taulukon toiminnallisuuden testaaminen on soveltuva testitapaus automatisointiin ja täytetään tarvittavat perustiedot. Testitapauksen erivaiheet kirjattiin seuraavanlaisesti:

Vaihe 1

- Testivaihe: Avataan yhden saatavan alitaulukko Laajenna valitut -painikkeella.
- Testitiedot: Valitse saatavat listalta yksi saatava ja laajenna valitut.
- Odotettu tulos: Saatavan valinta onnistuu ja saatavan alitaulukko avautuu.

Vaihe 2

- Testivaihe: Ensimmäisen vaiheen saatavan alitaulukko on avattuna. Avataan toisen saatavan alitaulukko samalla painikkeella.
- Testitiedot: Valitse toinen saatava listalta ja paina Laajenna valitut -painiketta.
- Odotettu tulos: Saatavan valinta onnistuu, Laajenna valitut -painike avaa uuden valitun saatavan alitaulukon ja ensimmäinen laajennettu alitaulukko pysyy myös auki.

Vaihe 3

- Testivaihe: Tiivistä vaiheen 1 ja 2 laajennetut alitaulukot.
- Testitiedot: Laajennetut alitaulukot ovat avattuina. Paina Tiivistä valitut -painiketta.
- Odotettu tulos: Tiivistä valitut -painike sulkee valittujen saatavien alitaulukot.

Vaihe 4

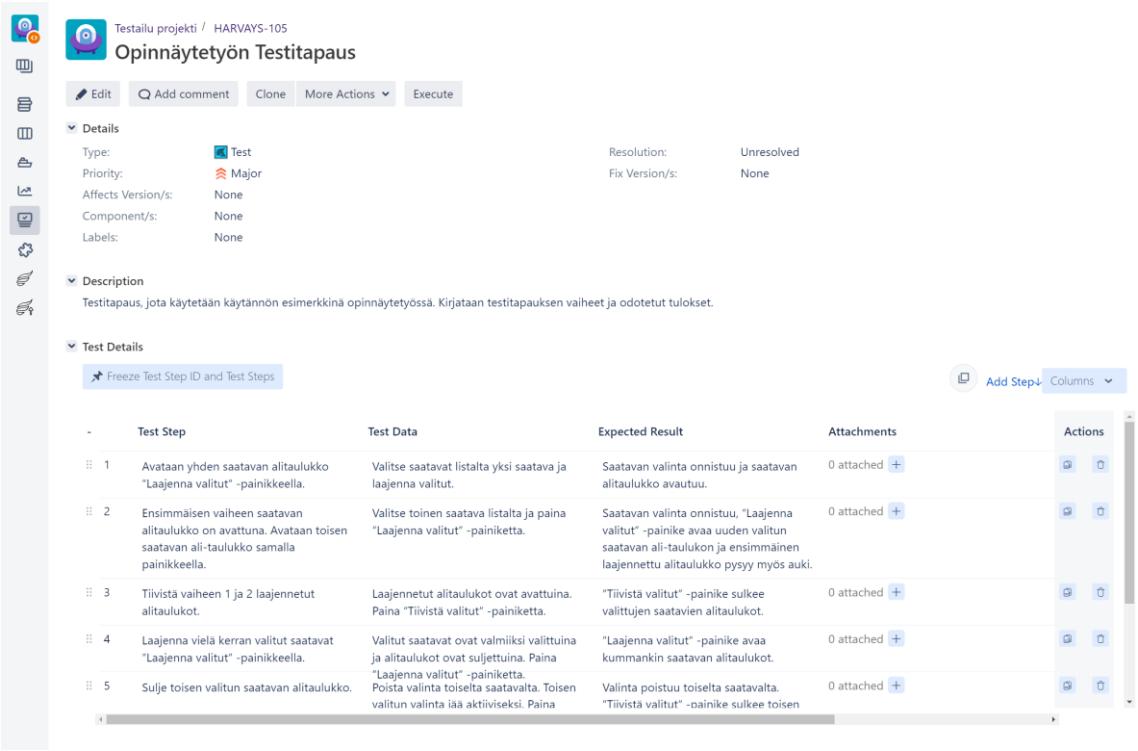
- Testivaihe: Laajenna vielä kerran valitut saatavat Laajenna valitut -painikkeella.
- Testitiedot: Valitut saatavat ovat valmiiksi valittuina ja alitaulukot ovat suljettuina. Paina Laajenna valitut -painiketta.

- Odotettu tulos: Laajenna valitut -painike avaa kummankin saatavan alitaulukot.

Vaihe 5

- Testivaihe: Sulje toisen valitun saatavan alitaulukko.
- Testitiedot: Poista valinta toiselta saatavalta. Toisen valitun valinta jää aktiiviseksi. Paina Tiivistä valitut -painiketta.
- Odotettu tulos: Valinta poistuu toiselta saatavalta. Tiivistä valitut -painike sulkee toisen valitun saatavan alitaulukon.

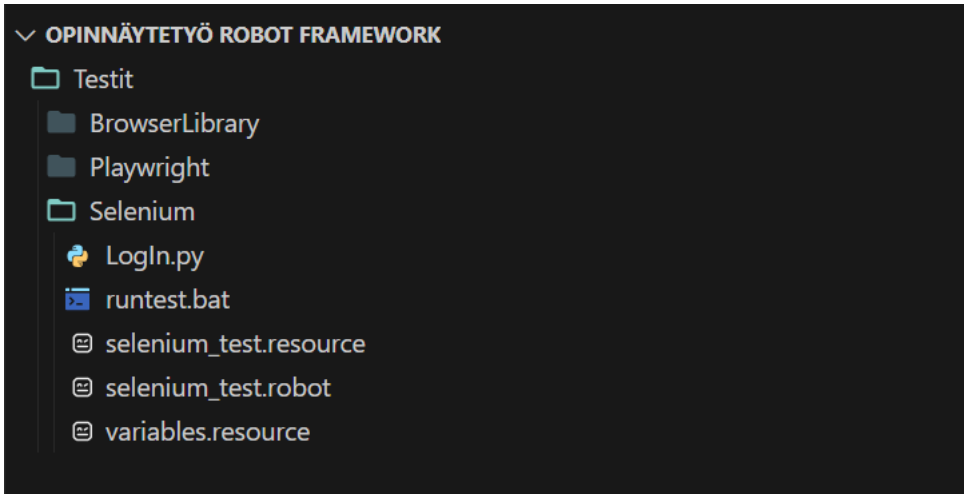
Loin testityyppisen työkohteen Jiraan ja kirjasin sinne edeltävät testivaiheet (kuva 23). Suunnitelma tuli valmiiksi ja oli aika siirtyä testitapauksen automatisointiin Visual Studio Codella.



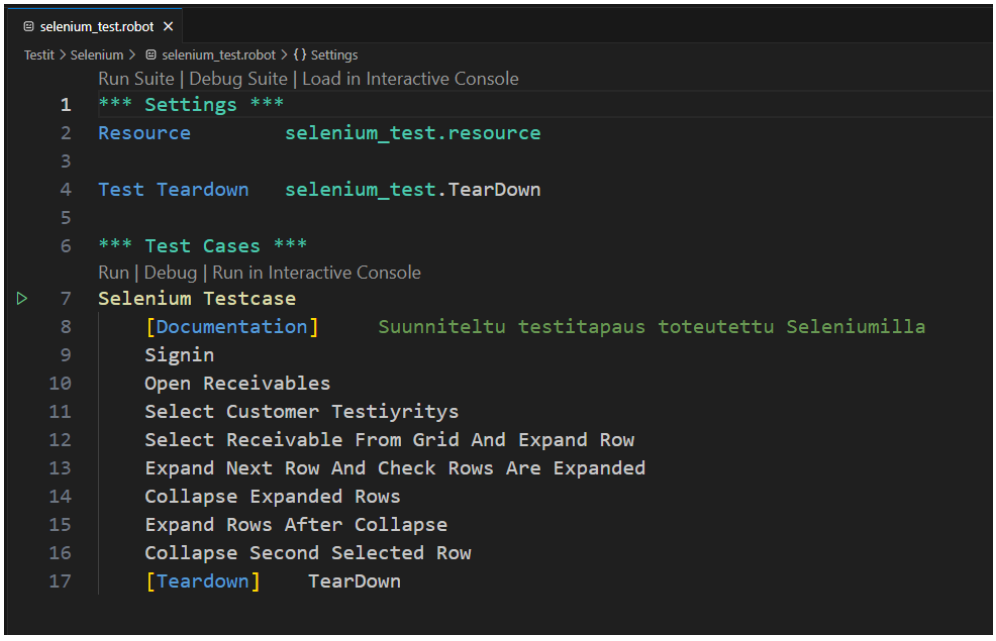
KUVA 23. Valmis testitapaus Jirassa

11 KÄYTTÖLIITTYMÄTESTI – SELENIUMLIBRARY

Aloitin testitapauksen automatisoinnin projektipohjaan, jonka tein valmisteluvaiheessa aiemmassa luvussa. Alussa suunnittelin tarvittavat tiedostot (kuva 24), joita tulen tarvitsemaan automaation toteutukseksi. Tein robot-päätteisen tiedoston varsinaisen testitapauksen rakenteen muodostamiseksi (kuva 25) ja resurssitiedoston testitapauksen avainsanojen ohjelmallista sisältöä varten. Asennettu SeleniumLibrary on liitetty resurssitiedostoon asettamalla asetukset -osioon **Library Browser** käyttämällä tabulointia.



KUVA 24. SeleniumLibrary-testitapauksen tiedostot



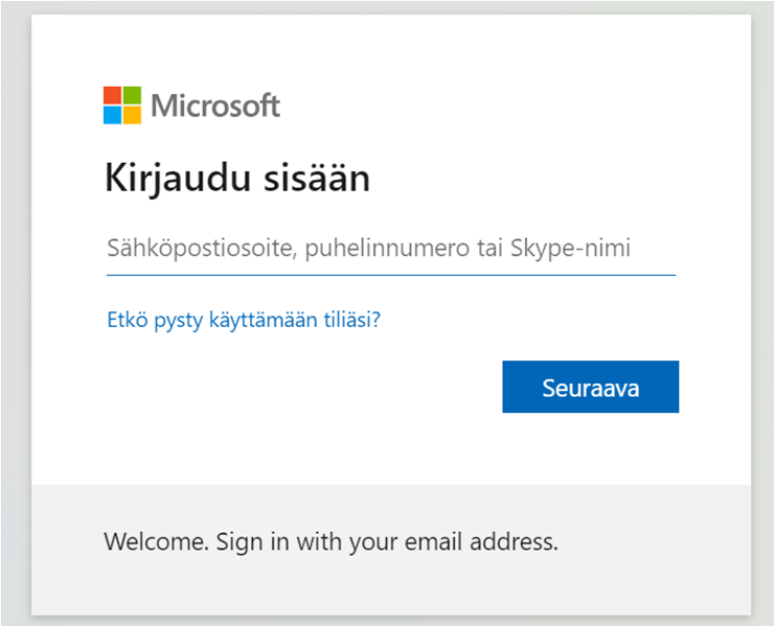
KUVA 25. SeleniumLibraryn testin rakenne robot -päätteisessä tiedostossa

Testitapauksen puolivaiheessa käsiteltiin ag-grid-elementtiä, jossa saatavat on listattu näkyville. Kuvan 26 funktioissa näytetään, miten taulukon rivejä ensin odotetaan siihen tarkoitetulla avainsanalla, jolle on annettu paikannin. Paikannin sisältää taulukolle ohjelmoidun attribuutin grid-id, jolla taulukon runko voidaan helposti kohdistaa avainsanalle. Paikantimelle annetaan myös rivin indeksiarvo, joka on arvoltaan numero. Tällä tavoin voidaan yksityiskohtaisesti kertoa ohjelmalle, mitä halutaan odottaa. Funktioissa klikataan myös rivin valinta painiketta (input), jolla valitaan rivit, mitä halutaan käsitellä ja ne laajennetaan laajennuspainikkeella. Tämän jälkeen avautuu rivin alitaulukko, jonka olemassaolo voidaan varmentaa testin myöhemmässä vaiheessa poimimalla rivistä id ja kirjoittamalla testiin avainsana, joka varmistaa alitaulukon läsnäolon. Set Suite Variable -avainsana asettaa muuttujan uudelleenkäytettäväksi myös toisille samassa tiedostossa oleville funktioille, jota kyseinen varmennus avainsana käyttää.

```
Select Receivable From Grid And Expand Row
    [Documentation]    Select row-index 1 from receivables grid and expand selected row
    Wait Until Element Is Visible    //div[@grid-id="receivablesGrid"]//div[@row-index=1]
    Click Element    //div[@grid-id="receivablesGrid"]//div[@row-index=1]//input
    ${first-row-id}=    Get Element Attribute    //div[@grid-id="receivablesGrid"]//div[@row-index=1]    row-id
    Set Suite Variable    ${first-row-id}
    Click Element    //button[@data-testid="receivablesExpandSelectedButton"]
    Wait Until Page Contains Element    //div[@row-id="detail_${first-row-id}"]

Load in Interactive Console
Expand Next Row And Check Rows Are Expanded
    [Documentation]    Select row-index 3 from receivables grid and expand selected row
    Wait Until Element Is Visible    //div[@grid-id="receivablesGrid"]//div[@row-index=3]
    Click Element    //div[@grid-id="receivablesGrid"]//div[@row-index=3]//input
    ${second-row-id}=    Get Element Attribute    //div[@grid-id="receivablesGrid"]//div[@row-index=3]    row-id
    Set Suite Variable    ${second-row-id}
    Click Element    //button[@data-testid="receivablesExpandSelectedButton"]
    Wait Until Page Contains Element    //div[@row-id="detail_${second-row-id}"]
```

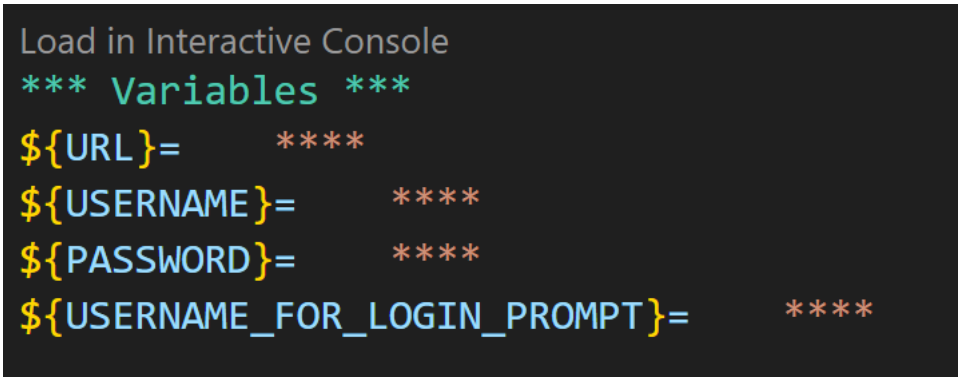
KUVA 26. Ag-grid-elementin käsittelyfunktiot SeleniumLibraryllä

Testattava sovellus on hyvin suojattu yrityksen toimesta, ja sivulle pääsee vain Azure-tunnuksilla kirjautumalla (kuva 27), joille on annettu käyttöoikeudet. Työntekijöillä on omat henkilökohtaiset käyttäjätilit, mutta testitapauksia varten on luotu omat testikäyttäjät, joilla on omat käyttöoikeudet päästä sivulle.



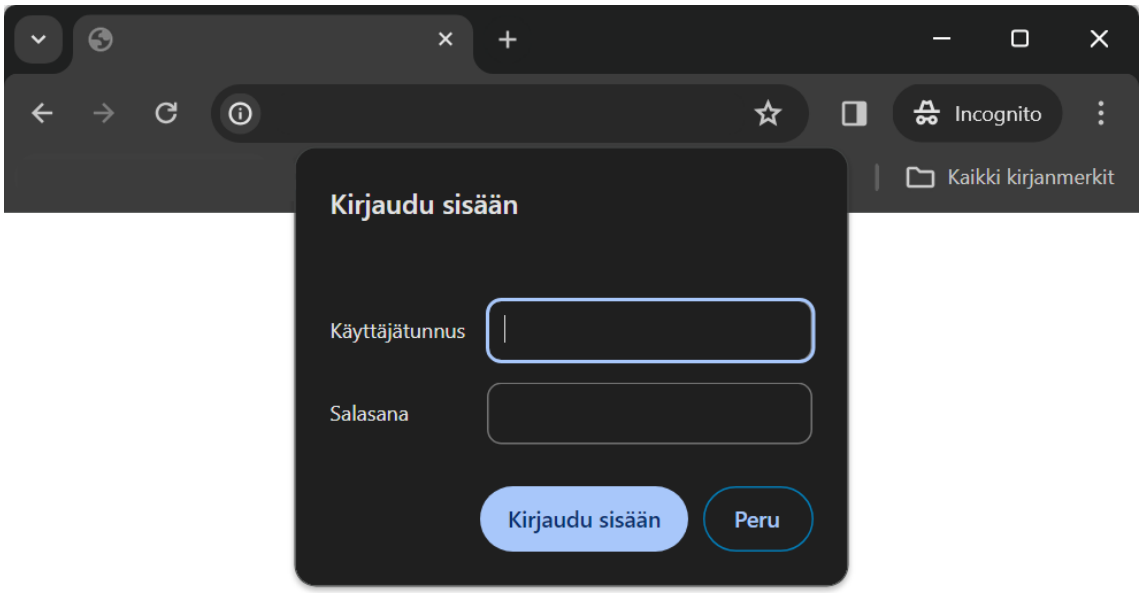
KUVA 27. Azuren kirjautumisikkuna

Tämän seurauksena testitapaus on suoritettava selaimen incognito-tilassa, jotta omat henkilökohtaiset tunnukset eivät ole selaimen välimuistissa eikä sivusto kirjaudu sisään automaattisesti testin aikana väärillä tunnuksilla. Lisäsin tästä syystä myös variables -nimisen resurssitiedoston, johon sisällytetään testikäyttäjän tunnus sekä salasana. Resurssitiedoston sisältö on sensuroitu. (Kuva 28)



KUVA 28. Variables -resurssitiedoston sisältö

Selain kysyy vielä yhden lisäaskeleen kirjautumisessa, jos välimuisti on tyhjä, jossa käyttäjä joutuu vielä syöttämään käyttäjätunnukset vielä kerran (kuva 29). Tämän toteuttaminen pelkästään SeleniumLibraryllä ei onnistu, koska tällä kirjautumisikkunalla ei ole olemassa paikantimia, joten jouduin käyttämään muita menetelmiä avuksi.



KUVA 29. Kirjautumispyyntö Chrome-selaimella

Ratkaisin ongelman rakentamalla avuksi Python-skriptin, joka simuloi käyttäjän tavoin kirjoittamista. Python-skripti ei vaadi paikantimen käyttöä Seleniumin avainsanan tavoin, vaan toteuttaa kirjoittamisen ilman sitä. Skripti kirjoittaa käyttäjänimen, painaa tabulaattoria, kirjoittaa salasanan ja painaa enter-painiketta. (Kuva 30)

LogIn.py

Testit > Selenium > LogIn.py > LogIn

```python
import pyautogui
import time

class LogIn:
    def __init__(self):
        # Optionally, initialize some variables or settings if needed
        pass

    def enter_credentials(self, email, password):
        """
        Writes an email, presses tab, writes a password, and presses enter.
        """
        # Wait a bit to ensure the correct input field is focused
        time.sleep(5)  # Adjust as necessary

        # Writes the email address
        pyautogui.write(email, interval=0.05)

        # Presses the Tab key
        pyautogui.press('tab')

        # Writes the password
        pyautogui.write(password, interval=0.05)

        # Presses the Enter key
        pyautogui.press('enter')
```

KUVA 30. Python-skripti kirjautumiseen

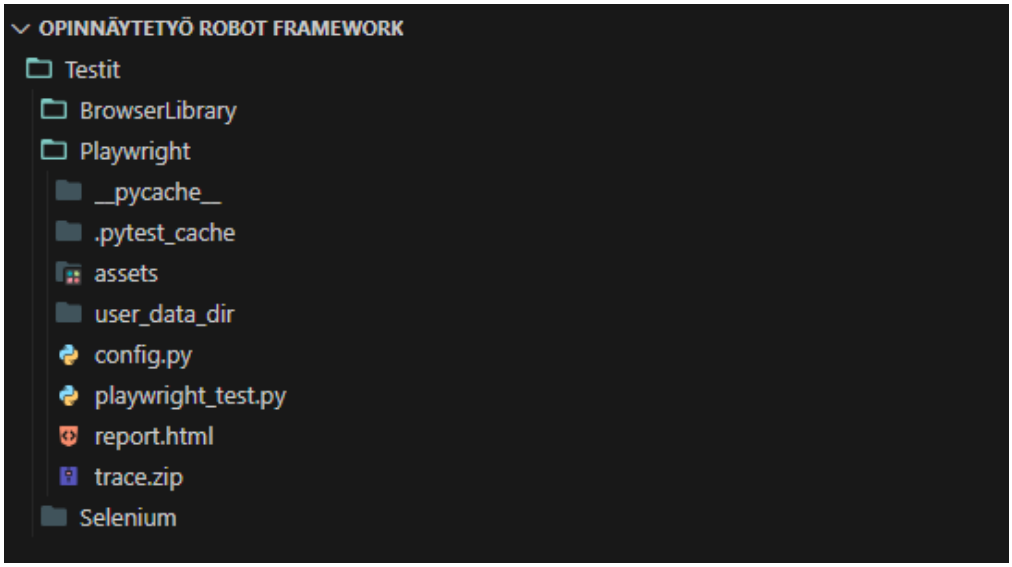
Skripti otettiin käyttöön testitapauksen kirjautumislogiikassa kohdassa Enter Credentials. Avainsana saa parametreikseen lyhennetyn käyttäjätunnuksen sekä salasanan variables-resurssitiedostosta. (Kuva 31)

```
Signin
    Open Browser    ${URL}    ${BROWSER}    options=add_argument("--incognito")
    Maximize Browser Window
    Set Selenium Speed  0.5
    Set Selenium Implicit Wait  10
    Set Selenium Timeout  10
    Wait Until Element Is Visible    //input[@id="i0116"]
    Input Text  //input[@id="i0116"]  ${Username}
    Press Keys    //input[@id="i0116"]    ENTER

    # Enter Credentials To Login Prompt
    Enter Credentials     ${Username_for_login_prompt}   ${Password}
    Wait Until Location Is    ${URL}/
```

KUVA 31. Testitapauksen kirjautumislogiikka

12 KÄYTTÖLIITTYMÄTESTI – PLAYWRIGHT

Jatkoin saman projektipohjan käyttöä, mutta tällä kertaa Playwrightille osoitettuun omaan kansioon (kuva 32). Valitsin Playwrightin tuetuista ohjelmointikielistä Pythonin, koska minulla oli sen käytöstä aiempaa kokemusta. Opiskelin Playwrightin testien käytäntöjä lähteen (29) eri osioista ja aloitin luomalla Python-päätteisen tiedoston nimeltä playwright_test, johon aloin rakentamaan samaa aiempaa testitapausta jäljitellen sitä, mitä tein aiemmin Seleniumissa.



KUVA 32. Playwright-testitapauksen tiedostot

Alkuun oli todella paljon vaikeuksia erilaisten ongelmien kanssa. Kirjautumisessa oli ongelmia Playwrightin valitseman vakioselaimen takia, joka oli Chromium. Chromium-selaimella sain kummallista yhteysvirhettä, jota en saanut aivan heti ratkaistua. Tutkin, miten voin vaihtaa käyttämääni selainta Playwrightissa ja vaihdoin sen käyttämään tavallista Chrome-selainta. Lisäsin myös metodin **new_context()**, joka luo uuden selainkontekstin. Selainkonteksti on eristetty ympäristö, jossa on omat evästeensä ja paikallinen tallennustila. Tämä on samanlainen kuin incognito- tai yksityinen selausikkuna. Kirjautumisongelma ratkesi näillä muutoksilla. Samaa ongelmaa ei ilmennyt ylimääräisen kirjautumispyynnön kanssa, mikä tapahtui Seleniumin testitapauksen yhteydessä.

Testiä kirjoittaessani laadin aluksi kaikki testitapauksen vaiheet yhteen funktioon, testasin sen toimivuuden, ja sen jälkeen muokkasin rakennetta järkevämmäksi siirtämällä vaiheet omiin funk-

tioihinsa. Tämä tekee testitapauksen eri osioista uudelleenkäytettäviä (kuva 33). Testiajon yhteydessä Playwright generoi projektin alle välimuistitiedostoja. Siirsin myös käyttäjätiedot ja verkko-osoitteen erilliseen Python -päätteiseen konfiguraatiotiedostoon.

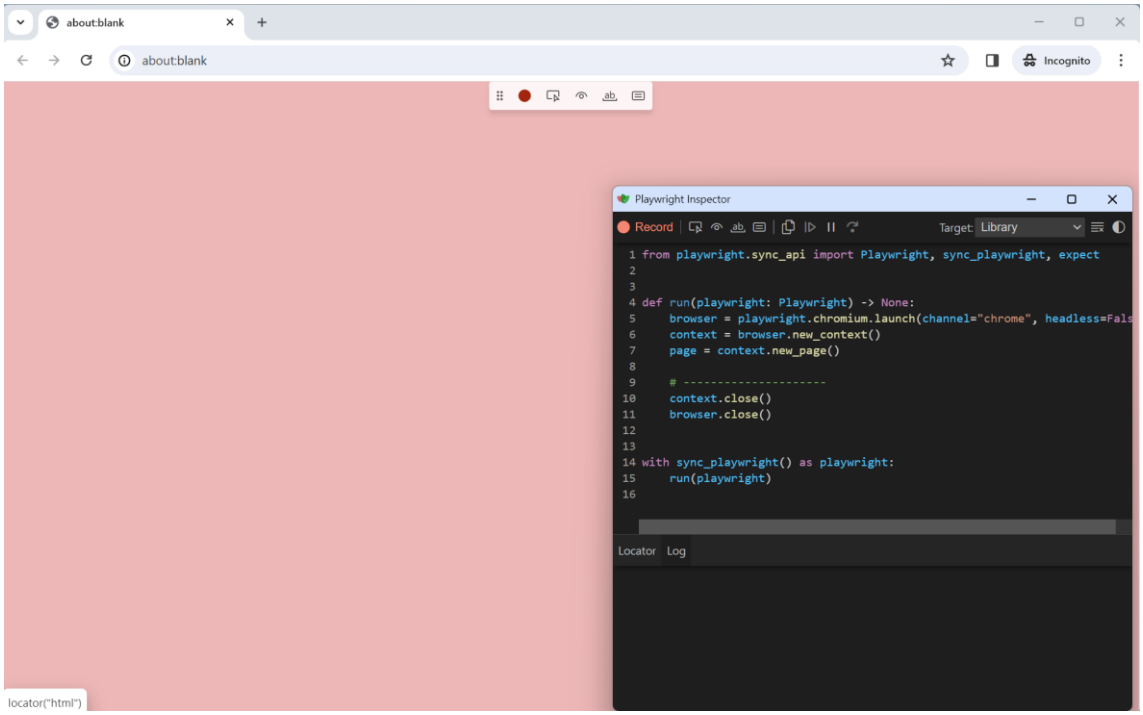
```python
def test_run(playwright: Playwright) -> None:
    browser = playwright.chromium.launch(headless=False, channel="chrome", args=["--start-maximized"])
    context = browser.new_context(no_viewport=True)

    # Start tracing before creating / navigating a page.
    context.tracing.start(screenshots=True, snapshots=True, sources=True)
    page = context.new_page()

    Signin(page)
    Open_Receivables(page)
    Select_Customer_Testiyritys(page)
    row_id_1 = Select_Receivable_From_Grid_And_Expand_Row(page)
    row_id_2 = Expand_Next_Row_And_Check_Rows_Are_Expanded(page, row_id_1)
    Collapse_Extended_Rows(page, row_id_1, row_id_2)
    Expand_Rows_After_Collapse(page, row_id_1, row_id_2)
    Collapse_Second_Selected_Row(page, row_id_2)

    # Stop tracing and export it into a zip archive
    context.tracing.stop(path="trace.zip")

    # ---------------------
    context.close()
    browser.close()
```

KUVA 33. Playwright testitapauksen pääfunktio, jossa on kerätyt testivaiheet

Playwrightin Codegen-työkalu osoittautui hyödylliseksi kehitysvaiheessa. Sen avulla sain hyvin paljon suuntaa siitä, miten testitapaukset loppujen lopuksi muotoutuvat. Työkalussa on paljon asetuksia Playwrightin tuetuille eri ohjelmointikielille ja se generoi käytettävää koodia sujuvasti nauhoittamalla käyttäjän toimintoja web-selaimessa (kuva 35). Käynnistin työkalun komennolla **playwright codegen --channel chrome**. Komennolle annetaan **channel** -vipu lisäksi, jotta se avaa selaimen käyttäjän haluamalla selaimella. Mukautin kuitenkin Inspectorin generoimaa koodia hieman haluamallani tavalla tunnistaa web-sovelluksen paikantimet. (Kuva 34)

```python
def Select_Receivable_From_Grid_And_Expand_Row(page):
    # Click the checkbox of the first row
    page.locator('//div[@grid-id="receivablesGrid"]//div[@row-index="1"]//input').click()

    # Fetch the 'row-id' attribute of the first row
    row_1 = page.locator('//div[@grid-id="receivablesGrid"]//div[@row-index="1"]')
    # Get the 'row-id' attribute
    row_id_1 = row_1.get_attribute("row-id")

    # Expand the selected row and verify that it is expanded
    page.get_by_test_id("receivablesExpandSelectedButton").click()
    expect(page.locator(f'//div[@row-id="detail_{row_id_1}"]')).to_be_visible()

    return row_id_1

def Expand_Next_Row_And_Check_Rows_Are_Expanded(page, row_id_1):
    # Click the checkbox of the second row
    page.locator('//div[@grid-id="receivablesGrid"]//div[@row-index="3"]//input').click()

    # Fetch the 'row-id' attribute of the third row
    row_2 = page.locator('//div[@grid-id="receivablesGrid"]//div[@row-index="3"]')
    # Get the 'row-id' attribute
    row_id_2 = row_2.get_attribute("row-id")

    # Expand again and verify that the rows are expanded
    page.get_by_test_id("receivablesExpandSelectedButton").click()
    expect(page.locator(f'//div[@row-id="detail_{row_id_1}"]')).to_be_visible()
    expect(page.locator(f'//div[@row-id="detail_{row_id_2}"]')).to_be_visible()

    return row_id_2
```

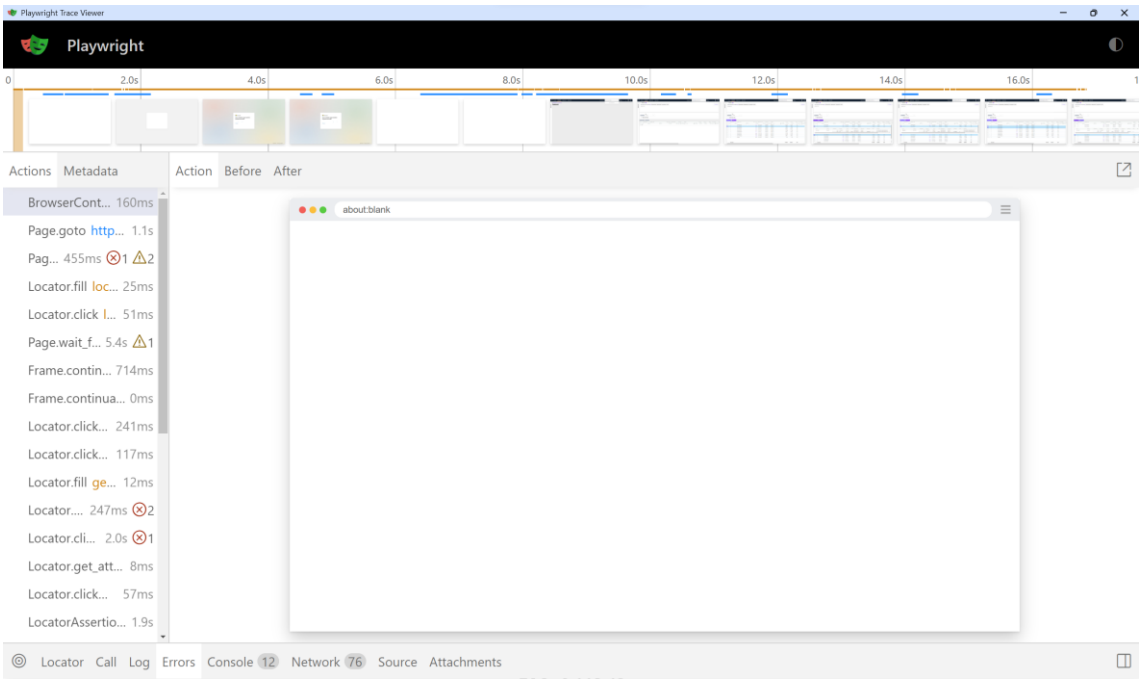
KUVA 34. Ag-grid-elementin käsittelyfunktiot Playwrightilla



KUVA 35. Playwright Codegen -työkalun Inspector-ikkuna sekä nauhoitettava selain

Valmiin testitapauksen jälkeen ei ollut ihan selkeää, miten Playwright toteuttaa testitulosten ulostulon tarkasteltavaksi. Lisäselvittelyjen jälkeen sain testitulokset näkyviin asentamalla erillisen Python ajoympäristön komennolla **pip install pytest** ja sen laajennuskirjaston **pip install pytest-html**. Tämän seurauksena testitapaus ajettiin **python playwright_test.py** komennon sijasta komennolla **pytest playwright_test.py --html=report.html**. Ajoympäristö mahdollisti testitapauksen testiraportin luomisen.

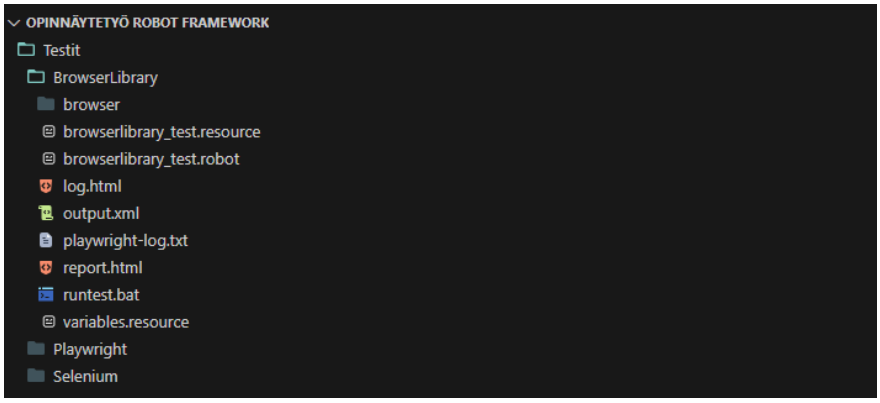
Tutustuin myös hyödylliseen työkaluun Trace, joka on hyödyllinen kehitysvaiheen seurannassa. Työkalussa nähdään täsmälleen, miten Playwright käyttäytyy testin jokaisella koodirivillä ja näyttää niistä valokuvat Trace Viewer -ikkunassa (kuva 36). Ikkunassa nähdään myös tarkasti, mitä elementtiä ajon aikana painettiin. Trace saatiin käyttöön lisäämällä pääfunktioon jäljityksen aloitus ja lopetus. Tulos saatiin avattua komennolla **playwright show-trace trace.zip**.



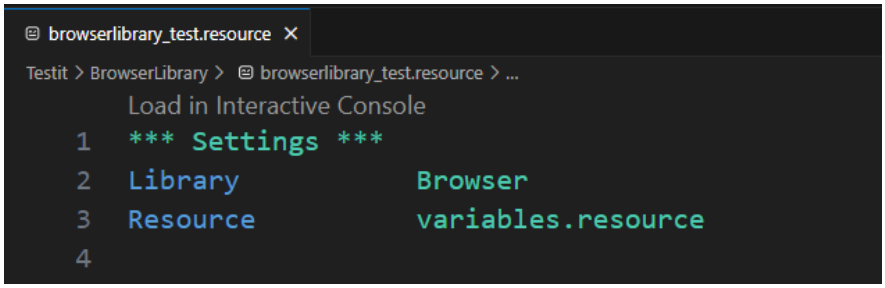
KUVA 36. Trace Viewer -ikkuna

13 KÄYTTÖLIITTYMÄTESTI – BROWSERLIBRARY

Testitapaus oli helppo aloittaa samankaltaisesti mitä Seleniumissakin, koska BrowserLibrary on vain liitettävä kirjasto Robot Frameworkin kehykseen. Loin robot- ja resource päätteiset tiedostot ja lisäsin muuttujatiedoston verkko-osoitetta sekä käyttäjätunnuksia varten, joita käytin resurssitiedoston ohjelmassa (kuva 37). Asennettu BrowserLibrary on liitetty browserlibrary_test resurssitiedostoon asettamalla asetukset osioon **Library Browser** käyttämällä väliin tabulointia (kuva 38).

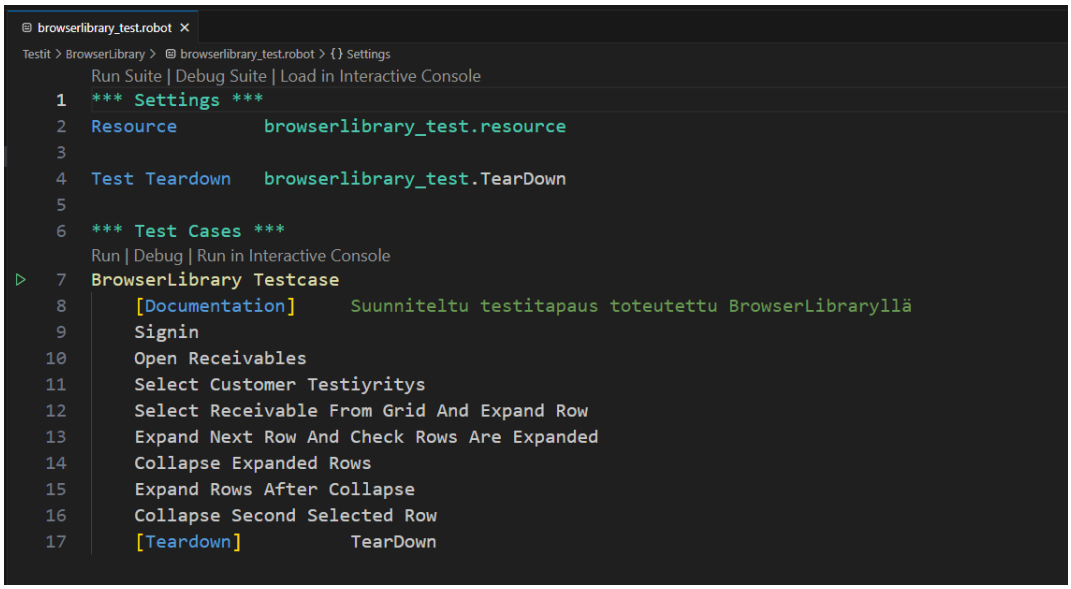


KUVA 37. BrowserLibrary-testitapauksen tiedostot



KUVA 38. BrowserLibraryn käyttöönotto Robot Frameworkissä

BrowserLibraryllä työskentely oli melkein täysin samanlaista mitä Seleniumissakin, mutta muutamilla muutoksilla. Testitapauksen rakenne ja kirjoitussäännöt menivät samalla periaatteella (kuva 39). Ainoastaan avainsanat muuttuivat erinimisiksi ja selaimen alustaminen. Testitapauksen kirjautumisfunktioon liitettiin **New Context** suljettu ajoympäristö, aivan kuten Playwrightissakin, ja sen toiminta oli samanlainen. Tämän seurauksena erillistä Incognito -tilaa ei tarvittu (kuva 40).



KUVA 39. BrowserLibrary testin rakenne -robot päätteisessä tiedostossa

```
Signin
    @{args}    Create List    --start-maximized
    New Browser    channel=chrome    headless=False    args=${args}
    New Context    viewport=${None}
    New Page    ${URL}
    Type Text    //input[@id="i0116"]    ${USERNAME}
    Click    //input[@id="idSIButton9"]
```

KUVA 40. BrowserLibraryn kirjautumisfunktio

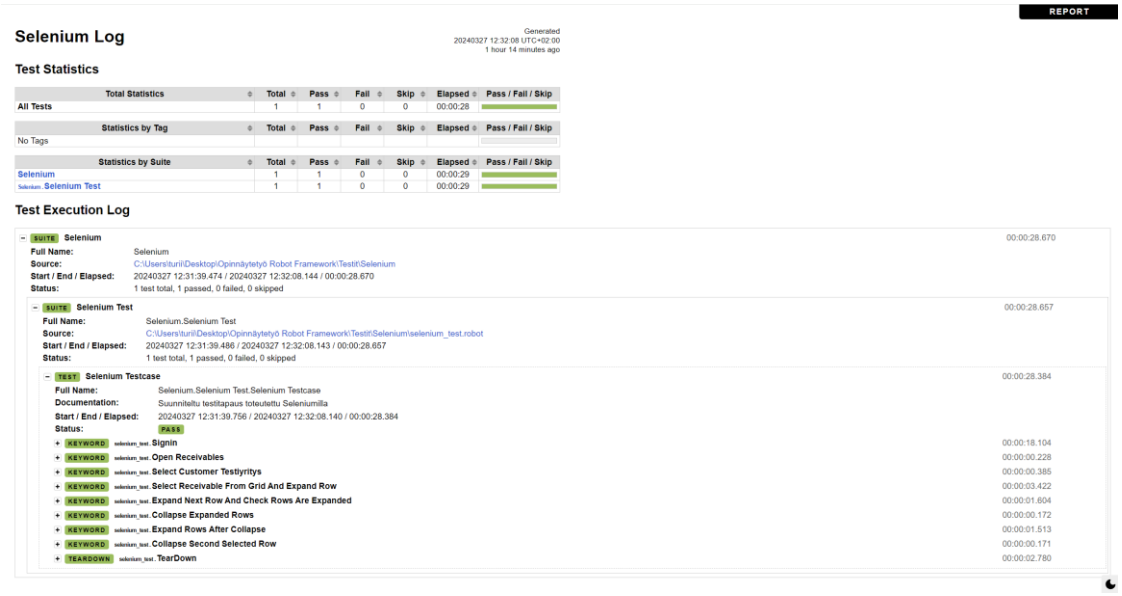
Selvitin, että erillisiä odotuksia ei BrowserLibraryllä juurikaan tarvita, vaan kirjasto hoitaa elementtien odottelut automaattisesti. Käytin odotusavainsanoja ainoastaan vain varmistaakseni, että elementti on näkyvillä tai piilotettuna testiajon hetkellä. (Kuva 41)

```
Select Receivable From Grid And Expand Row
    [Documentation]    Select row-index 1 from receivables grid and expand selected row
    Wait For Elements State    //div[@grid-id="receivablesGrid"]//div[@row-index=1]    visible
    Click    //div[@grid-id="receivablesGrid"]//div[@row-index=1]//input
    ${first-row-id}=    Get Attribute    //div[@grid-id="receivablesGrid"]//div[@row-index=1]    row-id
    Set Suite Variable    ${first-row-id}
    Click    //button[@data-testid="receivablesExpandSelectedButton"]
    Wait For Elements State    //div[@row-id="detail_${first-row-id}"]    visible

Load in Interactive Console
Expand Next Row And Check Rows Are Expanded
    [Documentation]    Select row-index 3 from receivables grid and expand selected row
    Wait For Elements State    //div[@grid-id="receivablesGrid"]//div[@row-index=3]    visible
    Click    //div[@grid-id="receivablesGrid"]//div[@row-index=3]//input
    ${second-row-id}=    Get Attribute    //div[@grid-id="receivablesGrid"]//div[@row-index=3]    row-id
    Set Suite Variable    ${second-row-id}
    Click    //button[@data-testid="receivablesExpandSelectedButton"]
    Wait For Elements State    //div[@row-id="detail_${second-row-id}"]    visible
```

KUVA 41. Ag-grid-elementin käsittelyfunktiot BrowserLibraryllä

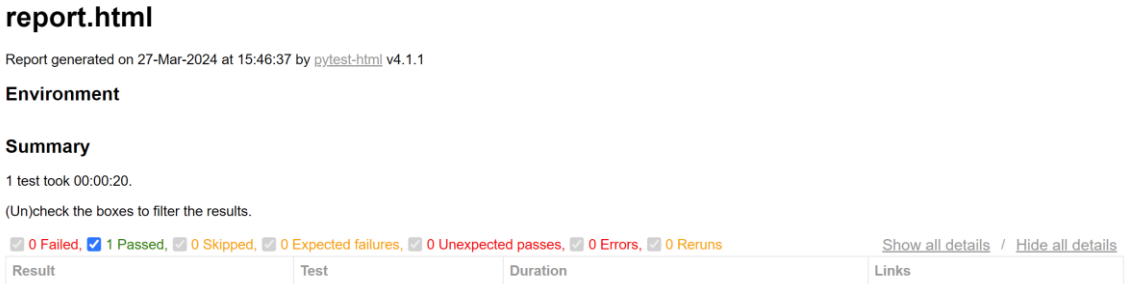
14 TESTITULOSTEN VERTAILU

Tämän luvun kuvissa nähdään testiautomaatiotulokset, jotka eri testiautomaatiokirjastot generoivat. Kuvassa 42 nähdään SeleniumLibraryn testiraportti. Seleniumin testiraportti sisältää erinomaisesti testiajon jokaisen vaiheen ja kuinka kauan vaiheen suorituksessa meni aikaa. Virhetapauksessa raportissa nähdään täsmälleen, millä rivillä testiajo on epäonnistunut. Virheen jäljitys on täten todella helppoa ja tehokasta, mikä tekee kirjaston käytöstä miellyttävää.



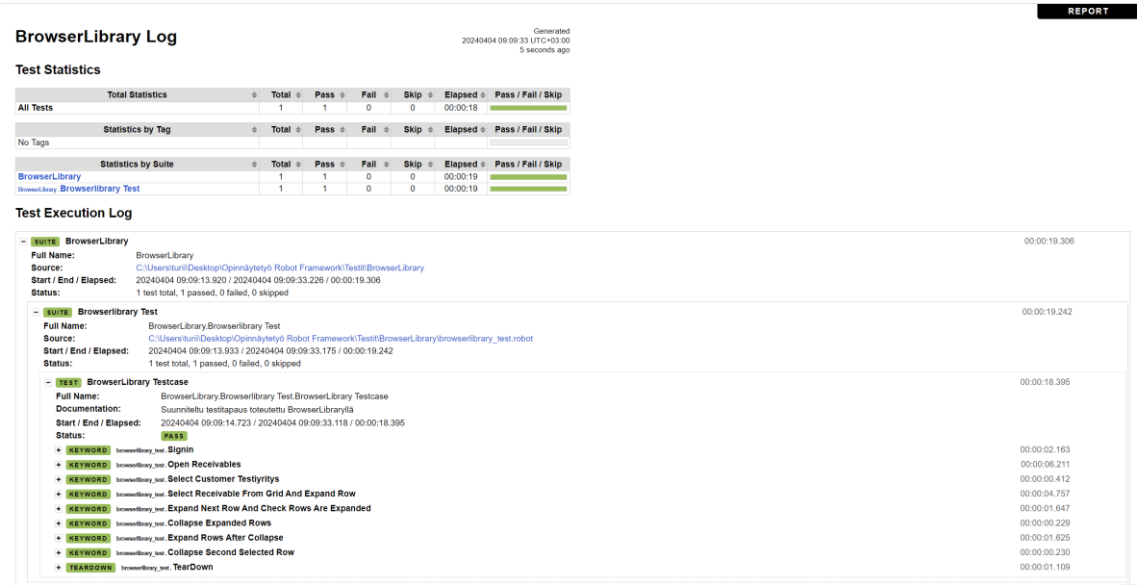
Kuva 42. SeleniumLibraryn testiraportti

Kuvassa 43 nähdään Playwrightin testiraportti. Itselleni ei kuitenkaan selvinnyt syytä siihen, miksi Playwrightin testiraportti ei näytä tarkasti testivaiheita millään tavalla. Näytä kaikki yksityiskohdat -painikkeesta huolimatta raportti ei anna yksityiskohtaista tietoa testeistä. Raportti kuitenkin näyttää testitapauksen suoritusajan sekä sen, menikö testitapaus läpi vai saiko se virheen ja kaatui.



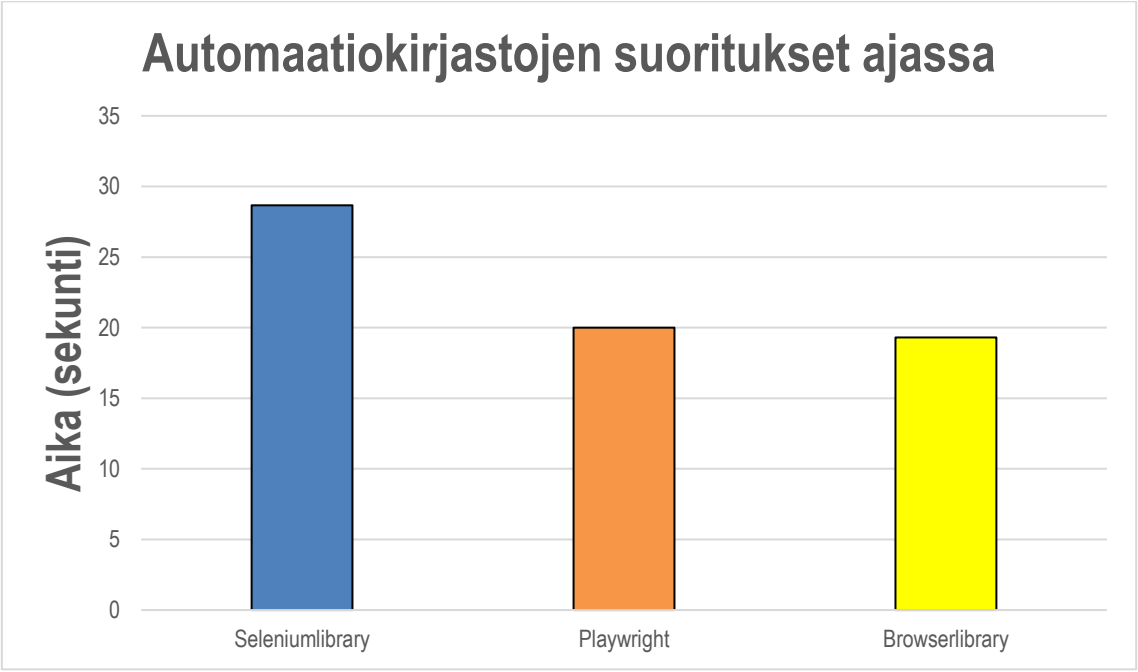
KUVA 43. Playwrightin testiraportti

Kuvassa 44 nähdään BrowserLibraryn generoima testiraportti. Se on käytännössä täysin sama kuin Seleniumissa.



KUVA 44. BrowserLibraryn testiraportti

Testiraporttien välillä voidaan tehdä vertailua. SeleniumLibrary suoritti testitapauksen 28,6 sekunnissa, Playwright tasan 20 sekunnissa ja BrowserLibrary 19,3 sekunnissa (kuva 45).



KUVA 45. Nopeuden vertailu havainnollistettu Excel-kaaviossa sekunneissa

Laskin, kuinka monta prosenttia nopeus kasvoi toisilla testiautomaatiokirjastoilla verrattuna SeleniumLibraryyn käyttäen kaavaa: **((uusi nopeus – vanha nopeus) / vanha nopeus) * 100**.

Tulokseksi saatiin Seleniumin ja Playwrightin välillä 30,2 prosentin muutos, mikä on huomattava parannus nopeudessa verrattuna SeleniumLibraryyn. BrowserLibrary suoriutui vielä hieman paremmin kuin Playwright, ja tulokseksi saatiin 32,6 prosentin muutos nopeudessa.

Testiraporttien ominaisuuksista voidaankin luoda vielä yhteenvetona taulukko, josta nähdään selvästi mitä ominaisuuksia ja puutteita raportit sisälsivät (kuva 46).

Ominaisuudet	SeleniumLibrary	Playwright	BrowserLibrary
Suoritusaika	X	X	X
Testivaiheet	X	X	X
Virheilmoitus	X	X	X
Helppolukuisuus	X		X

KUVA 46. Testiraporttien ominaisuudet

En onnistunut saamaan Playwrightin testiraportin testivaiheita näkyviin, mutta se on jollain tavalla mahdollista. Kehittäjän näkökulmasta testivaiheiden katselmointi testitapauksen virhetilanteessa on erinomainen lähde virheiden kohdentamiseen ja korjaamiseen testitapauksen koodissa. SeleniumLibraryn ja BrowserLibraryn testiraportit ovat helppolukuisia ja ilmoittavat tarpeelliset tiedot selkeästi. Lisäksi niiden tapa esittää testivaiheet on asianmukainen, eikä laajempaa selvitystyötä tarvita, jotta testivaiheet saadaan näkyviin.

15 JOHTOPÄÄTÖKSET

Aloittaessani web-testauksen ja tutustuessani testiautomaatioihin käytin SeleniumLibraryä, kuten yrityksessäni yleisesti oli tapana luoda testiautomaatioita kyseisellä kirjastolla. Tämä kokemus tietenkin edesauttoi opinnäytetyön Selenium-testitapausta, koska tein harjoittelujakson jo yrityksessä käyttäen Seleniumia työtehtävissäni. Selenium oli minulle silloin uusi, laajempi opiskeltava aihealue. Yllätyin aluksi, miten helposti ymmärrettävä Selenium oli ja kuinka nopeasti sitä pystyi oppimaan. Uutena tulokkaana testiautomaatioissa minulla ei ollut vielä paljoakaan tietoa muista testiautomaatioihin liittyvistä kirjastoista ennen opinnäytetyön tekemistä.

Varsinaisten testitapausten luonti Playwrightilla ja BrowserLibraryllä on testiraportin tulosten vertailun perusteella paljon tehokkaampaa kuin Seleniumilla luodut testitapaukset. Haastavinta Playwrightissa oli testitapauksen avainsanoja vastaavat ohjelmasyntaksit, jotka toimivat halutulla tavalla, samoin kuin ne toimivat Seleniumissa ja Browserissa. Playwrightin testitapaus toi kuitenkin hieman haasteita sen käytön kokemattomuuden vuoksi, mutta kokemuksen lisääntyessä myös käytöstä tuli suoraviivaisempaa.

Avainsanojen vastaavuusongelmaa ei juurikaan ollut BrowserLibraryssä, sillä samankaltaisuuksien vuoksi testitapauksen mukauttaminen BrowserLibrarylle oli mukavaa ja vaivatonta. BrowserLibraryn kehittäjille suunnattua dokumentointisivua katselmoidessa huomasin, että avainsanat Seleniumin testitapauksen kanssa eivät paljoakaan muutu.

Työn etenemistä nopeutti Jirassa luotu tarkka suunnitelma, jonne testitapauksen vaiheet kirjattiin tarkasti sekä saman testitapauksen uudelleenluonti toisen testiautomaatiokirjaston testitapauksessa oli helppo visioida jo valmiiksi etukäteen. Testitapausten onnistumisen kannalta luotettavuudessa ei ollut juurikaan ongelmia minkään työkalun kohdalla.

16 POHDINTA

Opinnäytetyössäni vertailtiin kolmea eri testiautomaatiokirjastoa web-sovellusten testaamiseen ja tutkittiin niiden hyötyjä ja mahdollisia haittoja. Työn käytännönosuudessa rakennettiin web-sovellukselle sama automaatiotesti jokaisella kolmella kirjastolla erikseen yhteen Visual Studio Coden projektikansioon. Pidimme yrityksessä joka viikko opinnäytetyöstäni viikkopalaveria yrityksessä valitun opinnäytetyönohjaajan kanssa työn sujuvuuden edistämiseksi.

Askel modernimpaan suuntaan testiautomaatioissa tapahtui käyttämällä BrowserLibraryä tai Playwright testiautomaatiokirjastoa. Näiden kahden suoriutuminen testeissä oli nopeampaa, mikä säästää yrityksiltä resursseja ja työtunteja testiautomaatioissa. Testien suoritusajat olivat niin merkittävästi nopeampia, että on suotavampaa käyttää Playwrightia tai BrowserLibraryä Seleniumin sijasta testiautomaatioiden optimointiin.

Vaikka Selenium on täysin toimiva tapa suorittaa web-sovellusten testiautomaatiot, on mielestäni tärkeää seurata vaikuttavia uusia trendejä ja seurata kehityksen kulkua alalla. Testitapauksissani ei selvinnyt mitään suurempaa ongelmaa, jonka vuoksi siirtyminen muihin testiautomaatiokirjastoihin ei olisi mahdollista. Mielestäni kehittäjille, jotka tuntevat SeleniumLibraryn jo ennestään, on helppo siirtyä käyttämään modernimpaa BrowserLibraryä yrityksessä pienellä vaivalla ja työmäärällä sekä jo olemassa olevien SeleniumLibrary testien korvaaminen BrowserLibraryllä onnistuisi myös suoraviivaisesti. Lisäksi liiketoiminnan näkökulmasta tulisi ainakin resurssien käytön osalta säästöjä.

Edellä mainitut perustelut ovat mielestäni jo riittävän hyvät valitsemaan BrowserLibraryn testiautomaatioiden rakentamiseen, mutta lopullinen päätös kirjaston käyttöönotosta jää yrityksen päätettäväksi.

LÄHTEET

1. Sommerville, Ian 2016. Software Engineering (10th edition). Global edition. Pearson.

2. Iivonen, Joonas. Testaussanasto – ohjelmistotestauksen tärkeimmät termit selitettynä Project-TOP. Hakupäivä 21.2.2024. https://projecttop.com/testaussanasto/.

3. GeeksforGeeks 2023. Test Plan – Software Testing. Hakupäivä 16.2.2024. https://www.geeksforgeeks.org/test-plan-software-testing/.

4. ProductPlan Jira. ProductPlan Glossary. Hakupäivä 21.2.2024. https://www.productplan.com/glossary/jira/.

5. Intellipaat What Is Jira (3.3.2023). Jira Explained in 3 minutes | Jira Tool For Beginners | Intellipaat. Saatavilla: https://www.youtube.com/watch?v=X2yYnADRYTw.

6. Testim 2019. What Is Test Automation? A Simple, Clear Introduction. Hakupäivä 21.2.2024. https://www.testim.io/blog/what-is-test-automation/.

7. ATR Soft (4.1.2028). Ohjelmistotestaus parantaa laatua. Hakupäivä 22.2.2024. https://www.atrsoft.com/teknologiat/ohjelmistotestaus-parantaa-laatua.

8. Mannotra, Vivek (9.6.2023). Guide to Web Application Testing. BrowserStack. Hakupäivä 22.2.2024. https://www.browserstack.com/guide/web-application-testing.

9. Mozilla Developer Network (muokattu 24.1.2024). What is JavaScript. Hakupäivä 23.2.2024 https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

10. Deremuk, Iryna (muokattu 23.1.2024). LitsLink. Web Application Architecture. Hakupäivä 23.2.2024 https://litslink.com/blog/web-application-architecture.

11. Schwartz, Chris (27.4.2022). Web Application Testing: The Basics of Web App Test. Hakupäivä 23.2.2024 https://www.leapwork.com/blog/web-application-testing-the-basics-of-web-app-test-automation.

12. Deepak, Gupta loginradius. 7 Common Web Application Security Threats. Hakupäivä 22.2.2024. https://www.loginradius.com/blog/identity/7-web-app-sec-threats/.

13. Hamilton, Thomas (muokattu 9.12.2023). Guru99. Web Application Testing: How to Test a Website. Hakupäivä 23.2.2024 https://www.guru99.com/web-application-testing.html.

14. Google for Developers (muokattu 15.2.2024). Backend Architectures for content-driven web app backends. Content-driven web applications. Hakupäivä 23.2.2024 https://developers.google.com/solutions/content-driven/backend/architecture.

15. Kasurinen, Jussi Pekka 2014. Ohjelmistotestauksen käsikirja. Docendo.

16. Valjas Opi Blogi (2.5.2019). Mitä integraatio, rajapinta ja api tarkoittavat? Hakupäivä 5.3.2024 https://valjas.fi/opi/blogi/mita-integraatio-rajapinta-ja-api-tarkoittavat/.

17. Postman 2024. API testing. What is API testing? Hakupäivä 5.3.2024 https://www.postman.com/api-platform/api-testing/.

18. Bakshi, Aseem (muokattu 22.9.2021) Webomates. Mixing UI and API Testing – Testing UI tricks. Hakupäivä 5.3.2024 https://www.webomates.com/blog/api-testing/the-value-of-mixing-ui-and-api-testing/.

19. Kumar, Pawan (9.6.2023) BrowserStack. Introduction to Robot Framework. Hakupäivä 6.3.2024 https://www.browserstack.com/guide/robot-framework-guide.

20. Robot Framework. Robot Framework User Guide Version 7.0. Hakupäivä 6.3.2024 https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

21. Robot Framework. SeleniumLibrary. Hakupäivä 6.3.2024 https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html.

22. Browser Stack. Selenium Testing. Hakupäivä 6.3.2024 https://www.browserstack.com/selenium.

23. Browser Library. Hakupäivä 6.3.2024 https://robotframework-browser.org/.

24. Robot Framework – Introduction (8.4.2021) 1.04 Browser Library. Saatavilla: https://www.youtube.com/watch?v=3BNVS6uiFeo&t=1s.

25. Lambdatest (17.3.2023). What Is Playwright? Playwright Testing Tutorial – A Guide With Examples Hakupäivä 8.3.2024 https://www.lambdatest.com/playwright.

26. Playwright. Fast and reliable end-to-end testing for modern web apps. Hakupäivä 8.3.2024 https://playwright.dev/.

27. Playwright. Generating Tests. Hakupäivä 8.3.2024 https://playwright.dev/docs/codegen-intro.

28. Playwright. Trace Viewer. Hakupäivä 8.3.2024 https://playwright.dev/docs/trace-viewer-intro.

29. Playwright. Installation. Hakupäivä 11.3.2024 https://playwright.dev/python/docs/intro.

30. Browser Library. Installation. Hakupäivä 13.3.2024 https://robotframework-browser.org/.