Tampereen ammattikorkeakoulu

Automatisoidun Playwright-testauksen käyttöönotto ohjelmistoprojekteissa

Aleksi Pynnönen

OPINNÄYTETYÖ
Kesäkuu 2025

Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

PYNNÖNEN, ALEKSI:
Automatisoidun Playwright-testauksen käyttöönotto ohjelmistoprojekteissa

Opinnäytetyö 31 sivua
Kesäkuu 2025

Opinnäytetyö sai alkunsa työelämän haasteista: Playwrightin käyttöönotto projektissa paljasti kompastuskiviä, osa koskien teknistä tasoa, mutta suuri osa syntyen mahdollisesta kitkasta ryhmädynamiikoissa, organisaatiorakenteissa ja ihmisten käyttäytymisessä. Tästä syntyi tavoite luoda dokumentti, joka helpottaisi Playwrightin käyttöönottoa tavalla, jolla testaus olisi arvoa lisäävä tekijä eikä pelkkää ajanhaaskausta.

Työn alkupuolella nojataan raskaammin alan kirjallisuuteen ensin ottamalla kosketusta verkkokäyttöliittymätestauksen kasvavaan rooliin sekä ohjelmistotestaukseen yleisesti. Tämän jälkeen aihe siirtyy ryhmädynamiikkoihin ja yksilöiden käyttäytymiseen tiimissä. Loppupuolisko dokumentista käsittelee Playwrightin sopivuutta eri kehitysympäristöissä, sen asennusta sekä testaukselle kriittisimpiä toimintoja ja yleisiä testinkirjoituskäytäntöjä.

Teknisiin esteisiin dokumentti tarjoaa jokseenkin suoria mutta ympäristöagnostisia ratkaisuja. Ihmis- ja organisaatiotasolla ongelmiin ei helppoja ratkaisuja ole, mutta niiden esille ottaminen ja käsittely antaa mahdollisuuden huomata vastaavat ongelmat omassa kehitystiimissä ajoissa.

Asiasanat: ohjelmistotestaus, automatisaatio, playwright

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Production

PYNNÖNEN, ALEKSI:
Implementing Automated Playwright Testing in a Software Project

Bachelor's thesis 31 pages
June 2025

The objective of the thesis was to create a document that would act as a quick start guide to adopting automated Playwright testing in a software development project. Automated testing is vital for ensuring the reliability of the software while keeping down the time spent on testing. The thesis covers human factors, such as motivation and how testing integrates into a development workflow. It also details the process of integrating Playwright into a development environment and writing tests, while maintaining platform agnosticism.

The thesis relied most heavily on subject-specific literature for its human factors sections. For the technical aspects, it primarily used official documentation, distilling it to the core elements of Playwright test writing.

While the thesis stayed in its scope, it was clear that the aspect of human factors goes much deeper than initially expected. The coverage on the topic was enough to avoid many of the most common pitfalls but only scratches the surface of the subject.

Key words: software testing, automatization, playwright

SISÄLLYS

TERMIT

CI/CD-putki	Automatisoitu prosessiketju, joka tukee jatkuvaa integraatiota (CI) ja jatkuvaa toimitusta (CD) esimerkiksi ajamalla automaattisesti testejä uusiin ohjelmistoversioihin
Kuvakaappaus	Kuva ruudusta, englanniksi ”screen capture”
Paikannin (Playwright)	Paikantimet (engl. locators) ovat Playwright-metodeja, jotka antavat Playwright-testien paikantaa elementtejä verkkosivulla
Playwright	Web-testauksen automaatiotyökalu
Ohjelmointiajapinta	Application programming interface eli API
Regressiovirhe	Ilmiö, jossa aiemmin toiminut toiminto lakkaa toimimasta uuden muutoksen seurauksena
SaaS	Liiketoimintamalli, jossa ohjelmisto tarjotaan asiakkaalle palveluna, usein pilvipalveluna.
Tilannekuva	Tallenne järjestelmän tai sovelluksen tilasta tietyllä ajanhetkellä. Käytetään yleisesti ohjelmistotestauksessa vertailupohjana muutoksille, esimerkiksi visuaalisissa testeissä. Englanniksi snapshot
Toiminto (Playwright)	Playwright-toiminnot (engl. actions) antavat Playwright-testien käyttää paikantimien löytämiä elementtejä
Verkkokäyttöliittymä	Web-teknologioilla (HTML, CSS, JavaScript) toteutettu käyttöliittymä selaimessa tai selainpohjaisessa sovelluksessa

1 JOHDANTO

Nykypäivän IT-teollisuudessa monet sovellukset ovat siirtymässä verkkokäyttöliittymiin. Samalla älylaitteet ovat yleistymässä niin työ- kuin kuluttajakäytössä ja älylaitteiden käyttö- ja kokoluokat ovat monipuolistumassa.

Tämä on johtanut tarpeeseen testityökaluille, joilla verkkokäyttöliittymiä voidaan testata helposti erilaisilla selaimilla ja laiteko'oilla. Tässä tarpeessa Playwright on noussut merkittävään asemaan nykypäiväisenä testaustyökaluna. Työkalun avulla kehittäjät voivat kirjoittaa nopeasti niin käyttöliittymän kuin visuaalisen eheydenkin varmentavia testejä.

Opinnäytetyön inspiraation taustalla on oma ensikokemus Playwrightin käyttöönotosta työelämässä. Dokumentin tavoite on olla lähtökohta Playwrightin käyttöönotolle kehitystiimissä. Käytännössä tämä kattaa sekä Playwrightin teknisen käyttöönoton että tiimityöhön liittyvät psykologiset näkökulmat, kuten yleisimpien inhimillisten virheiden välttämisen sekä sujuvan yhteistyön tukemisen.

Opinnäytetyö ei käsittele jokaista Playwrightin toimintoa eikä tule antamaan suoria ohjeita usealle kehitysympäristölle. Yleiset vaihtoehdot Playwrightin kehitysympäristöön integrointiin tullaan mainitsemaan, mutta esimerkeissä pidättäydytään yksinkertaiseen esimerkkiympäristöön ja Playwrightin suosimaan TypeScript-kieleen. Opinnäytetyössä on käytetty tekoälyä muutaman kappaleen kirjoitus- ja kielioppivirheisiin oikolukuvaiheessa, mutta tutkittu tieto on hankittu ilman tekoälyä.

2 OHJELMISTOTESTAUS

2.1 Ohjelmistotestauksen asema

Ohjelmistotestaus on ohjelmistokehityksen keskeinen, mutta laajalti huomiotta jäävä ja usein huonosti hoidettu vaihe. Testaus saattaa jäädä taka-alalle aikataulu- ja budjettipaineiden vuoksi, vaikka sen merkitys laadun varmistamisessa voi olla kriittinen. Erityisesti testausvaiheen resursointi ja suunnittelu jäävät usein riittämättömiksi, mikä johtaa tuotteen toiminnallisiin puutteisiin ja käyttäjäkokemuksen heikkenemiseen.

Testauksessa usein erityisen haasteelliseksi koettu osa-alue on ollut automatisoitu käyttöliittymätestaus. Sitä on pitkään pidetty epäluotettavana ja työläänä verrattuna muihin testausmuotoihin. Tähän on vaikuttanut muun muassa käyttöliittymäalustojen standardien puute ja testien suuri herkkyys käyttöliittymämuutoksille. Tilanne on kuitenkin viime vuosina alkanut hitaasti muuttumaan ja paikoitellen Playwrightia saatetaan jopa pitää trendikkäänä vaihtoehtona.

2.2 Testauksen aloittamisen vaikeus ja miksi se silti kannattaa

”Move fast and break things” -ajattelu on käytännössä Piilaakson motto, ja ajan käyttö testien kirjoittamiseen tuntuu helposti ajan haaskaukselta, varsinkin kun tehtävälistalla odottaa jo seuraava ohjelmiston toiminto. Uusien toimintojen koodauksesta saa myös enemmän dopamiinia ja yläpeukkua työkavereilta kuin testien kirjoittelusta.

Näissä tunnelmissa on kaikille helppo unohtaa kokonaan ohjelmiston testaus, joka voi joskus olla koko tiimille tuntematon aihe. Joskus uutta tuotetta kirjoitettaessa se voi myös olla tuotteen tulevaisuuden kannalta oikea päätös. Kasvavassa palvelumuotoisessa ympäristössä pitkän tähtäimen suunnittelu on välttämätöntä. Automatisoitu testaus puolestaan tarjoaa käytännöllisen ja kestävän ratkaisun, joka kannattaa ottaa huomioon osana tätä suunnittelua.

Jossakin vaiheessa projekti saapuu vaiheeseen, jossa testejä pitäisi kirjoittaa, mutta koska valtaosa tiimistä ei ole tottunut kirjoittamaan testejä, jää aihe ikuisesti roikkumaan, ellei joku tiimistä ota riskiä ärsyttävästi puuttumalla koko ryhmän työskentelytapoihin. Ihmisten ollessa sosiaalisia eläimiä tämä tietenkin harvemmin tapahtuu.

Abstraktit keskustelut kaikille tutuista teoreettisista hyödyistä voivat saada osakseen hyväksyviä nyökkäilyjä, mutta jos ylhäältä ei tule käskyä aloittaa testien kirjoittamista, ei muutosta todennäköisimmin pelkällä puheella tule. Ehkä yksi tärkeimmistä asioista testauksen myynnissä muulle tiimille on todistaa, että testaus ei ole se työmäärähirviö, jonka ohjelmistokehittäjät ovat siitä mielikuvituksissaan luoneet. Ylemmille portaille organisaatiossa usein perustelut keskittyvät enemmän arvon lisäämiseen.

2.3 Miksi Playwright?

Verkkokäyttöliittymät ovat yleistymässä miltei kaikissa tekniikan alan tuotoksissa, usein jopa konteksteissa, joissa käyttäjä ei ymmärrä käyttävänsä niitä. Monet puhelinsovellukset ovat verkkonäkymiä, ja ohjelmat tietokoneilla ovat kasvavissa määrin Electron-sovelluksia eli käyttävät Chromiumia verkkokäyttöliittymän näyttämiseen. Jopa SpaceX:n Dragon-avaruusaluksen käyttöliittymä rakennettiin Chromium-selaimen päälle (Hnaide, 2020).

Syitä verkkokäyttöliittymien yleistymiselle on monia, mutta keskeisin on ehkä selainten monien laaja tuki eri käyttöjärjestelmillä, joka tekee verkkokäyttöliittymistä käytännössä alusta-agnostisia. Jokaiselle alustalle natiivin sovelluksen kehityksen sijasta on helppo hyödyntää verkkoselaimia, jotka jo antavat ohjelmistokehyksen yhtenäisen käyttöliittymän tuottamiseen kaikille alustoille.

Monet käyttöliittymät ovat muuttumassa entistä monimutkaisemmiksi, kun vaatimukset kasvavat ja ohjelmistot pysyvät pidempään jatkuvassa kehityksessä Software as a Service (SaaS) -jakelumallin seurauksena. Markkinoiden keskittyessä tämän jakelumallin suuntaan tulee ohjelmiston luotettavuudesta varsinkin yritysmarkkinoilla entistä kriittisempi tekijä.

Playwright on käytettävissä kaikilla suosituilla käyttöjärjestelmillä, ja sillä voi testata käyttöliittymiä Chromiumilla, Firefoxilla sekä Webkitillä. Testeillä voi siis käytännössä kattaa koko verkkokäyttöliittymämarkkinat. Testien tekeminen onnistuu useilla eri kielillä, ja ne voidaan suorittaa automatisoidusti ja päällekkäin integraatiota ja toimitusta testaavissa CI/CD-putkissa. Testit eivät myöskään tarvitse selainkohtaista koodia. Playwrightin dokumentaatio on korkealaatuista ja sen käyttäjien määrä on nopeassa kasvussa, mikä todennäköisesti tulee johtamaan jo kattavien yhteisöresurssien määrän ja laadun kasvuun.

3 KOHTUUDEN RAJAT JA TESTAUKSEN AJATUSMAAILMA

3.1 Testauksen yleinen lähtöpiste

Ohjelmistotestien kirjoittamisen aloittaminen, varsinkin ns. kylmiltä, jättää ajattelun usein asennemaailmaan, jossa testeillä yritetään todistaa, että ohjelma toimii. Tämä on kuitenkin käytännöllisyyden kannalta aikaa haaskaava tapa, ja psykologisella tasolla varma tapa ampua omaa tuotteliaisuutta ja motivaatiota jalkaan. (Myers ym, 2012, s. 5–6.)

Luvun tavoite on uudistaa lukijalle testauksen metodologiaa ja sanastoa tehokkaampaan suuntaan. Samalla ohjaten testausta kriittisimpien ohjelmistovirheiden löytämiseen ja pois itseä sabotoivista lähestymistavoista.

3.2 Käytännölliset rajoitteet

Ohjelman virheettömyyden todistaminen testaamalla on käytännössä mahdotonta sillä testauksen monimutkaisuus kasvaa eksponentiaalisesti ohjelman koon kasvaessa. Eli mitä monimutkaisempi ohjelma on testattavissa, sitä lähemmäs tullaan pistettä, jossa maailmankaikkeuden lämpökuolema tulee vastaan ennen ohjelman täydellistä testaamista. Tämän mittakaavan testaukseen voi luonnollisesti olla vaikea saada budjettia. (Myers ym, 2012, s. 7.)

Koska testausresurssit ovat käytännössä aina rajalliset, täydellisen ohjelman todistaminen mahdotonta. Siksi testejä on pakko rajata ja priorisoida. Hyvä lähestymistapa on kääntää ajattelutapa ympäri ja ohjelmiston oikein toimivuuden sijaan alkaa aktiivisesti etsimään virheitä.

3.3 Psykologiset rajoitteet

Ajattelutavan muuttaminen on myös psykologisesti tuottoisampaa. The Art of Software Testing-kirjassa (Myers ym, 2012) annetaan esimerkki, jossa henkilö,

jonka oletetaan ratkaisevan ristisanakirjan vartissa, saa todennäköisimmin vähemmän aikaan kuin henkilö, jonka tehtävä on vain ratkaista mahdollisimman monta sanaa. (Myers ym, 2012, s. 6–7.)

Tämä sanoilla leikkiminen ei myöskään rajoitu pelkästään tavoitteen määritykseen, vaan koskee myös sanastoa testien löytäessä virheen, eli kun testi löytää virheen, sitä yleensä kutsutaan epäonnistumiseksi, vaikka testi onnistui tarkoitetussa työssään oikeaoppisen sanaston perusteella. Kirjassa tätä virheellisenä epäonnistumisena pidettävää ajattelutapaa verrataan terveydenhuollon alan laboratoriotesteihin, joissa sanastoa käytetään oikein. Jos lääkäri passittaa potilaan testeihin ja testeissä ei löydy oireiden syytä, eivät testit tällöin löytäneet potilaan tautia, ja ne voidaan tällöin nähdä epäonnistuneina. Koska ohjelmiston kanssa voi olettaa, että mikään ohjelma ei ole virheetön, voidaan sitä myös oletusarvoisesti kohdella ”sairaana potilaana”, eli testi, joka löytää virheen, on onnistunut testi. (Myers ym, 2012, s. 6–7.)

Tämä päivitetty sanasto helpottaa myös tuoteomistajien, asiakkaiden tai muiden esimiesten kanssa kommunikointia ja neuvottelua, sekä suojaa ryhmähenkeä tarpeettomilta kolhuilta. Virheen olemassaolo tietenkin nähdään jossain määrin aina ”epäonnistumisena”, mutta pyöräyttämällä sanasto ympäri ja ilmaisemalla, että ”testi onnistui löytämään virheen”, aikataulumuutokset ymmärretään helpommin kaikilla osapuolilla. (Myers ym, 2012, s. 6–7.)

3.4 Virheiden löytäminen

Tehokkaan testauksen tavoitteena on löytää kriittiset virheet silloin, kun niitä esiintyy. Tämän vuoksi testauksen tulisi olla kohdistettua ja harkittua. Koska koko ohjelmaa ei voida testata kattavasti, hyvä lähtöpiste on tunnistaa ohjelmiston toiminnan kannalta kriittisimmät osa-alueet.

Kartoituksen jälkeen testejä kirjoitettaessa toimivuus on luonnollisesti ensimmäinen vaihe, jota testataan. Testien tulisi kuitenkin myös sisältää tarkistuksia, jotka varmistavat, ettei ohjelmisto tee mitään oletusten vastaista, jos käyttäjä ei seuraa

”kehittäjän maalaisjärkeä”. Mitä esimerkiksi tapahtuu, jos päivämääräkenttään annetaan tekstiä tai jos käyttäjä palaa edelliselle sivulle täyttäessään lomaketta? Rajatapauksia tulisi huomioida mahdollisimman kattavasti.

Testausprosessin edetessä virheitä tulee todennäköisesti jossain vaiheessa vastaan. Virheen löytyessä ei kuitenkaan ole suositeltavaa siirtyä heti seuraavaan osaan sovellusta, vaan kannattaa keskittyä tarkemmin siihen kohtaan, josta virhe löytyi. Virheet nimittäin ilmenevät usein ryppäinä, ja yhden korjaaminen saattaa paljastaa lisää piileviä ongelmia. (Myers ym, 2012, s. 170.)

Virheitä korjattaessa kannattaa myös huomioida, minkä luontoisia virheet ovat, kuka ne oli tehnyt ja miten virhe olisi voitu välttää. Tällä tavalla virheitä voidaan välttää tulevaisuudessa. Virheen tekijöistä ei kuitenkaan tule tehdä syntipukkia sillä tavoite on vähentää virheiden määrää eikä syyllistää henkilöä. (Myers ym, 2012, s. 172.)

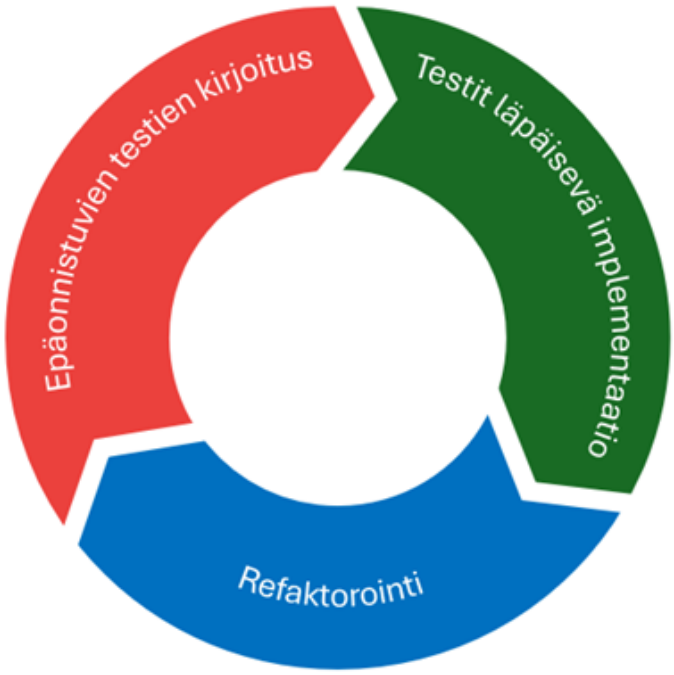
4 TESTIVETOINEN KEHITYS

Ohjelmistokehityksen alalla on monenlaisia kehitystapoja ja erilaiset tiimit voivat soveltaa erilaisia kehitystapoja eri konteksteissa. Aiheen vuoksi tämä luku käsittelee testivetoisen kehityksen lähestymistapaa. Tämä toivottavasti antaa eväät joko joidenkin käytäntöjen lainaamiseen tai testivetoisuuden laajempaan käyttöönottoon tai antaa lähtökohdan testauksen soveltamiseen muissa kehitystavoissa. Luvun tavoite ei kuitenkaan ole käännyttää ketään testivetoiseen kehitykseen, vaan käyttää sitä kärjistettynä esimerkkinä.

Testivetoisessa kehityksessä käytäntö on, nimensä mukaisesti, ohjata kehitystä kirjoittamalla testejä, eli esimerkiksi testit voidaan kirjoittaa ennen koodia ja koodi kirjoitetaan sen jälkeen läpäisemään nämä testit. Tavoitteena on, että testit kattavat mahdollisimman suuren osan koodin toiminnallisuudesta ja että testien kirjoittaminen integroituu osaksi kehitysprosessia sen sijaan, että se jäisi tekemättä tai lykkääntyisi kiireessä viime hetkelle.

Testivetoinen koodaus voi myös johtaa modulaarisempaan projektikoodiin, koska testivetoisesti koodattaessa on modulaarisuus usein luonteva ratkaisu, ja voi rajatussa määrin vähentää erillisen laadunvalvojan tarvetta. Prosessi soveltuu myös hyvin parikoodaamiseen sen keskustelua ja päätöksentekoa ohjaavan rakenteen vuoksi. (Irvine, 2023, s. 22.)

Tunnettu prosessi testivetoisessa koodauksessa on *punainen-vihreä-refaktorointi*-työnkulku, joka asettaa kehitysprosessin perinteiset vaiheet uuteen järjestykseen ja aloittaa testien kirjoittamisen ennen uuden toiminnon koodausta. Työnkulku on nimensä mukainen kolmivaiheinen syklinen prosessi, joka luonnollisesti alkaa punaisesta vaiheesta. Kuvio 1 esittelee prosessin visuaalisesti aiheelle tyypillisessä syklikaaviossa.



Kuvio 1. *Punainen-vihreä-refaktorointi*-syklikaavio.

Punainen-vihreä-refaktorointi-työnkulku soveltuu moneen eri kehitysprosessiin. Vaikka sen voi helposti yhdistää yksikkötestaukseen, soveltuu se myös läpivientitestaukseen. Tosin testien monimutkaisuus saattaa tietyissä konteksteissa kasvaa oletettua suuremmaksi, mikä puolestaan hidastaa iteroinnin nopeutta.

Punaisessa vaiheessa aloitetaan toiminnon koodaaminen kirjoittamalla testi jo ennen toiminnon luomista. Testin ei tarvitse olla monimutkainen. Ensimmäisellä kierroksella testi voi vain tarkistaa onko, uutta elementtiä edes olemassa. Elementti voi olla jotain yksinkertaista, kuten kellonaikakenttä lomakkeeseen. (Irvine, 2023, s. 22–32.)

Vihreässä vaiheessa lisätään vain toiminnallisuus, jolla punaisen vaiheen testi menee läpi. Tarkoitus ei siis ole kirjoittaa koko toimintoa. Sivulle lisätään vain käyttöliittymään lomake, jossa on kellonaikakenttä. Vaiheen tulee tehdä mahdollisimman vähän tämän saavuttamiseen. Esimerkiksi palvelinpuolisen tiedonkäsittelykoodin lisääminen on tämän kierroksen laajuuden ulkopuolella. (Irvine, 2023, s. 22–32.)

Refaktoroinnin pitäisi olla kaikille koodareille tuttu, eli tässä vaiheessa siistitään koodi sisäisesti ja riippuen käytännöistä lisäillään visuaaliset muutokset, kuten

CSS-tyylittelyt. Myös mahdolliset kehitysympäristön antamat varoitukset kuuluisi siivota pois, jos mahdollista. Tavoite on saada joka kierroksella jotain ”valmiiksi” ja pitää jokaisen kierroksen tavoite kohtuullisena. Määrittelyn laajentuminen testin kirjoituksen jälkeen tulee johtamaan vähemmän kattaviin testeihin ja virheiden yleistymiseen. (Irvine, 2023, s. 22–32.)

Refaktoroinnin jälkeen palataan taas punaiseen vaiheeseen. Toisella kierroksella voi vaikka kirjoittaa testin, joka yrittää lähettää lomakkeen palvelimelle. Testin kirjoituksen jälkeen koodataan lomakkeen palvelimelle lähetys, ja niin edespäin. Helpoimmillaan testit voivat olla yksinkertaisia testejä esimerkiksi tekstikentälle ja painikkeille, mutta monimutkaisen lomakkeen kanssa testin koko saattaa paisua ilman hyvää testisuunnitelmaa. (Irvine, 2023, s. 22–32.)

5 PLAYWRIGHTIN KÄYTTÖÖNOTTO JA ASENTAMINEN

5.1 Ympäristökohtaiset valinnat

Playwrightin voi asentaa monella eri tavalla riippuen ympäristössä. Kielen valinta on projekti- ja tiimikohtaista esimerkiksi jos projekti käyttää jo monessa kontekstissa Javaa, voi tällöin Playwrightin kirjoitus Javalla olla paras lähestymistapa.

Tässä dokumentissa käsitellään vain TypeScript-version käyttöönottoa, sillä Playwright on kirjoitettu TypeScriptillä, joka on täten sen ”natiivi” kieli ja yleisesti nähdään Playwrightin oletuskielenä. Tämän takia TypeScript-versio myös saa uudet toiminnot ensimmäisenä. Testiajurivalinta seuraa samaa linjaa, eli testiajurina toimii Playwrightin virallinen testiajuri. Eri vaihtoehdot ovat listattuina alla taulukoissa 1 ja 2.

Taulukko 1. Testiajurit. (Playwright. 2025a; Playwright. 2025b; Playwright. 2025c; Playwright Community. 2022; Vitest. 2025; yyvess. 2023.)

Kieli	**Testiajurit**
JavaScript/ TypeScript	Playwright Test, Jest, Mocha, Vitest
Python	Pytest
Java	JUnit, TestNG
C#	MSTest, NUnit, xUnit

Taulukko 2. Playwrightin tukemat kielet.

Kieli	**Hyödyt**	**Haitat**
JavaScript/ TypeScript	Ensisijainen ja natiivi kieli Saa päivitykset ensimmäisenä	NPM-riippuvuudet
Python	Pytest-integraatio, yksinkertainen syntaksi, laadunvarmistajille tuttu kieli	Hitaammat päivitykset
Java	Integraatio yritysympäristöissä	Hitaammat päivitykset

Kieli	Hyödyt	Haitat
		Monimutkaisempi käyttöönotto
C#	Integraatio yritysympäristöissä	Hitaammat päivitykset Monimutkaisempi käyttöönotto

5.2 Asentaminen

TypeScript-versio Playwrightista löytyy Node.js-pakettina NPM:stä. Alla olevat komennot asentavat Playwrightin, selaimet testien ajoa varten sekä Playwrightin virallisen testiajurin.

```
npm init -y
npm install --save-dev playwright
npx playwright install
npm install --save-dev @playwright/test
```

Playwrightin asentamisen jälkeen voi asennuksen toimivuuden varmistaa luomalla ensimmäisen testin komennolla ”`npx playwright codegen https://tuni.fi`”. Komennon ajaminen avaa selainnäkymän, jossa elementtien klikkaaminen lisää kontekstuaalisia testejä samalla auenneeseen Playwright Inspector-ikkunaan. Koodin voi tämän jälkeen kopioida ja tallentaa projektissa sijaintiin ”`/tests/esimerkki.spec.ts`”.

Testin voi tämän jälkeen ajaa komennolla ”`npx playwright test`”. Playwright-testien suorittaminen saattaa vahingossa ajaa myös muita JavaScript- ja TypeScript-tiedostoja kuin Playwrightin testitiedostot. Tämän estäminen käsitellään konfiguraatiotiedostossa luvussa 5.3. Lisäämällä komentoon ”`--ui`”-valitsimen saa testien ajamiseen myös graafisen käyttöliittymän.

5.3 Konfiguraatiotiedosto

5.3.1 Konfiguraatiotiedoston aloittaminen

Asentamisen jälkeen tulee säätää konfiguraatiotiedosto. Kaikkia asetuksia ei tarvitse ymmärtää tai hyödyntää heti, mutta nopeaa alkua varten tiedostoon kannattaa sisällyttää ainakin tiettyä alustavaa koodia, niin sanottua ”boilerplatea”. Luvun 5.3 alaluvuissa (5.3.1–5.3.5) esitellään konfiguraatiotiedoston sisältö vaiheittain. Näistä koodiesimerkeistä muodostuva kokonaisuus on koottu valmiiksi konfiguraatiotiedostoksi, joka löytyy liitteistä helposti kopioitavassa muodossa.

Alla oleva koodipätkä rajaa testien haun tiettyyn kansioon ajaen vain ”.spec.ts”-päätteisiä tiedostoja ja antaa helpon paikan säätää aikakatkaisun pituutta sekä testien käyttöliittymän päälle ja pois vaihtamista. Koodi toimii muiden asetusten lisäämiselle runkona. 5.3 Ala-lukujen (5.3.1–5.3.5) käsittelemät lisäykset lisätään ”defineConfigin” sisälle.

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
    testDir: 'tests',
    testMatch: ['*.spec.ts'],
    timeout: 30_000,
    use: {
        headless: true,
    },
});
```

5.3.2 Selainten lisäämien

Oletusarvoisesti Playwright ajaa testit vain Chromiumilla työpöytänäkymässä. Jokainen oletusasetusta laajempi selainkonfiguraatio lisätään omana projektinaan konfiguraatiotiedostoon. Selaimia voi lisätä myös erilaisina laiteprofiileina, kuten mobiililaitteen näytönkokoa ja asetuksia emuloivina versioina.

Jotta Playwrightia voi hyödyntää täysimääräisesti, voi mukaan lisätä Firefox- ja Safari-testit alla olevalla koodilla. Playwrightin ”`deviceDescriptors-Source.json`”-tiedostosta löytyvät myös ohjeet testien ajamiseen erilaisten mobiilinäkymien kautta (Microsoft, 2025).

```
projects: [
    {
        name: 'chromium', use: { ...devices['Desktop Chrome'] },
    }, {
        name: 'firefox', use: { ...devices['Desktop Firefox'] },
    }, {
        name: 'webkit', use: { ...devices['Desktop Safari'] },
    },
],
```

5.3.3 Raportterit

Tässä vaiheessa Playwright voi ajaa testejä jo useammalla selaimella. Seuraava luonnollinen etappi on siis testitulosten raportointi – eli miten ja mihin testien tulokset tallennetaan ja esitetään. Playwright tuottaa raportit niin sanottujen raporttereiden avulla. Taulukossa 3 listattujen raporttereiden lisäksi Playwright tarjoaa myös ohjelmointirajapinnan omien raporttereiden toteuttamiseen.

Taulukko 3. Playwrightin raportterit.

Raportteri	Avainsana	Käyttö
List-raportteri	list	Esittää komentorivikäyttöliittymässä listan testeistä
Line-raportteri	line	Esittää testien edistyksen komentorivikäyttöliittymässä yhdellä rivillä
Dot-raportteri	dot	Esittää jokaisen testin tilanteen komentorivikäyttöliittymässä värillisenä pisteenä
HTML-raportteri	html	Tallentaa tulokset HTML-tiedostona

Raportteri	Avainsana	Käyttö
JSON-raportteri	json	Tallentaa tulokset JSON-tiedostona
Blob-raportteri	blob	Tallentaa tulokset .zip-tiedostoon, joka sisältää yksirivisen JSON-tiedoston
JUnit-raportteri	junit	Tallentaa tulokset JUnit-tyylisenä XML-tiedostona
GitHub Actions-annotaatiot	github	Tuottaa komentorivikäyttöliittymään annotaatioita Githubin toiminnoille kuten CI/CD-putkille

Konfiguraatioon voidaan esimerkkiä varten lisätä *lista*, *JSON*- ja *HTML*-raportterit. HTML-raportteri voidaan myös asettaa automaattisesti selaimessa avautuvaksi tilanteissa, joissa joku testeistä epäonnistuu.

```
reporter: [
    ['list'],
    ['json', {  outputFile: 'results.json' }],
    ['html', { open: 'on-failure' }],
],
```

5.3.4 Tilannekuva- ja kuvankaappaustestit konfiguraatiossa

Playwright voi ottaa testien ohessa tilannekuvia ja kuvankaappauksia, joita se voi verrata aikaisempiin näytteisiin tarkistaakseen onko testien välillä tullut muutoksia. Toiminnon testikäyttöpuolta käsitellään luvussa 6.4, mutta konfiguraatiomuutokset voidaan lisätä jo tässä vaiheessa. Alla olevista asetuksista ”`maxDiffPixels`” määrittää, kuinka monta pikseliä näytteet voivat erota toisistaan, ja ”`threshold`” määrittää, kuinka paljon värit voivat erota näytteiden välillä.

```
expect: {
    toHaveScreenshot: {
        threshold: 0.1,
        maxDiffPixels: 10
    },
```

```
    toMatchSnapshot: {
        threshold: 0.1,
        maxDiffPixels: 10
    },
},
```

Kuvanäytteiden testauksessa on suositeltavaa, että vertailukuvana käytettävä referenssikuva otetaan mahdollisimman samankaltaisella laitteisto- ja ohjelmistokonfiguraatiolla kuin testien ajamiseen käytettävä järjestelmä. Selkokielisemmin sanottuna kannattaa varmistaa, että vertailukuvia otettaessa laitteella on mieluiten sama käyttöjärjestelmä, komponentit ja ohjelmistoversiot. Kannettavalla tietokoneella testailtaessa voi myös laturiin kytkeminen vaikuttaa tuloksiin.

5.3.5 webServer

WebServer-konfiguraatiota ei ole lisätty esimerkkikonfiguraatioon, jotta esimerkki säilyisi mahdollisimman yleiskäyttöisenä ja sovellettavissa erilaisiin kehitysympäristöihin. WebServer-ominaisuus on kuitenkin erityisen hyödyllinen tapauksissa, joissa testattavan sovelluksen tarvitsee testejä varten käynnistää taustalle sovellusrajapintoja tai mikropalveluita.

WebServer on määritelty objektitaulukkona, jossa jokainen objekti edustaa omaa palvelintaan. Alla olevassa esimerkissä käynnistetään taustapalvelin portissa 8080, jos ympäristömuuttujissa kerrotaan, että olemassa olevaa palvelinta ei ole jo saatavilla. Olemassa olevan palvelimen käytön määrityksestä voi olla hyötyä esimerkiksi, jos kehitysympäristössä testien halutaan käyttävän jo käynnissä olevaa palvelinta, mutta CI/CD-putkessa Playwrightin oletetaan käynnistävän uusi palvelin testiä varten.

```
webServer: [
  {
    command: 'npm run start:backend',
    port: 8080,
    url: 'http://localhost:8080/backend',
```

```
    timeout: 30000,
    reuseExistingServer: !process.env.CI //Ympäristömuuttuja
  }
]
```

6 PLAYWRIGHTILLA KOODAUS

6.1 Koodigeneraattorin ja manuaalisen koodaamisen rooli

Tässä luvussa päästään vihdoin testien koodaamiseen. Alkeellisten testien luonti on kuitenkin onneksi suhteellisen yksinkertaista, ja Playwrightin koodigeneraattorilla kokeileminen antaa jo yleisen käsitteen testien kirjoittamisesta. Alla oleva komento avaa Chromium-selaimen sekä ikkunan, johon generoitu koodi ilmestyy.

```
npx playwright codegen https://tuni.fi
```

Lyhyen käytännön kokeilun jälkeen lukijan tulisi hallita Playwrightin perustoiminnot siinä määrin, että dokumentin loppuosaa voidaan käyttää yksittäisten testirivien toiminnan havainnollistamiseen. Koodigeneraattorin tuottama koodi ei kuitenkaan ole parasta mahdollista eikä välttämättä kata kaikkia tärkeitä toimintoja. Generaattori on silti hyödyllinen työkalu testien rungon rakentamisessa, jonka yksityiskohtia voi jälkikäteen tarkentaa. Playwright Inspector-ikkunassa tulee valita ympäristölle sopiva testiajuri. Luvun 5 esittämää asennusprosessia seurattaessa oikea ajuri on jo oletuksena valittu *Test Runner*.

6.2 Sivujen lataaminen

Generoidun testin alussa löytyy todennäköisimmin ”`page.goto('/')`”-metodi. Mitä koodi ei kuitenkaan näytä, on metodin ”`waitUntil`”-asetus, jonka oletusarvo on ”`load`”. Oletusarvoa seuratessa testi jää odottamaan selaimelta ”load”-tapahtumaa, joka ilmenee vasta, kun sivu on ladannut kaikki CSS-tyylittelyt, JavaScriptit ja kuvat.

Tämä on kuitenkin esimerkiksi sivulla linkin klikkaamista varten täysin tarpeetonta. Nopeampia vaihtoehtoja ovat ”`commit`”, joka jatkaa testiä heti, kun palvelimen viesti on vastaanotettu ja dokumentin lataus alkanut, sekä ”`domcon-`

tentloaded”, joka jatkaa testiä heti, kun DOM:in lataus on valmistunut. Esimerkiksi seuraava koodirivi käynnistää sivunlatauksen ja jatkaa testiä heti, kun DOM on valmis.

```
await page.goto('/', { waitUntil: 'domcontentloaded' });
```

6.3 Paikantimet, toiminnot ja sisällön tarkistaminen

Paikantimia käytetään Playwrightissa elementtien valitsemiseen. Paikantimet voivat valita elementtejä niiden roolin, tekstisisällön, tunnisteiden ja muiden niihin liitettyjen tietokenttien mukaisesti. Esimerkiksi ”tallenna”-painikkeen voisi löytää koodaamalla ”`await page.getByRole('button', { name: 'Tallenna' });`”. Esimerkin ”name”-parametrin kirjainkoolla on väliä, mutta käyttämällä säännöllistä lauseketta, tässä tilanteessa ”/tallenna/i”, voi elementtejä myös etsiä välittämättä kirjainkoosta.

Paikantimet ovat tarkkoja, eli paikannin tuottaa virheen, jos se löytää useamman kuin yhden elementin. Paikantimia käytettäessä on suositeltavaa hakea elementtejä rooli- ja tekstipohjaisesti. ID- ja CSS-paikannusta tulee välttää, jotta testejä ei tarvitse uudelleenkirjoittaa esimerkiksi sivun asettelun päivittyessä.

”Toiminnot” antavat testin vaikuttaa valittuihin elementteihin muun muassa klikkaamalla, täyttämällä tekstikenttiä tai valitsemalla valikon vaihtoehtoja. Edellisen esimerkin tallennuspainiketta voisi esimerkiksi klikata vaihtonäppäin pohjassa lisäämällä koodin perään ”`.click({ modifiers: ['Shift'] });`”

```
page.getByRole('button', { name: 'Tallenna' }).click({ modifiers: ['Shift'] });
```

Lomakkeita täytettäessä tulee usein käytettyä metodeja kuten ”`.fill('Mikko');`” tekstikenttien täyttämiseen tai ”`.selectOption({ label: 'polkupyörä' });`” listatuista vaihtoehdoista valitsemiseen. Sivun sisältöä puolestaan testataan ”assert”-lausekkeilla, joilla voi tarkistaa, löytyykö sivulta haluttu

elementti tai muu piirre. Alla oleva esimerkkikoodi tarkistaa, onko sivulla otsikko nimellä ”Tampereen yliopisto”.

```
await expect(page.locator('h1')).toContainText('Tampereen yliopisto');
```

Jos ”expect”-lauseke epäonnistuu annetussa tehtävässä kuten elementin löytämisessä, testi pysähtyy, ellei testi käytä ”`expect.soft`”-metodia. Tällöin epäonnistuminen merkitään ja testin ajo jatkuu. Metodin käyttö voi johtaa helpompaan vianetsintään, jos sama virhe vaikuttaa esimerkiksi useampaan elementtiin. Harvinaisissa tapauksissa se voi myös välttää esimerkiksi tilanteen, jossa yksi virhe piilottaa toisen virheen, jonka piilossa olo puolestaan johtaa sprintin väärinarviointiin.

6.4 Kuvankaappaukset ja tilannekuvat

Tähän mennessä testit ovat vain kattaneet elementtien olemassaolon tarkistuksen, mutta tilannekuvilla voi myös automatisoidusti löytää sivun asetteluun vaikuttavat regressiovirheet. Idea on ihmiselle intuitiivinen. Testejä varten otetaan kontrollikuvat testattavista sivuista, joita verrataan uusissa testeissä otettuihin kuviin. Uudet tilannekuvat ja kuvankaappaukset uusia testejä ja muuttuneita sivuja varten otetaan ajamalla alla oleva komento.

```
npx playwright test --update-snapshots
```

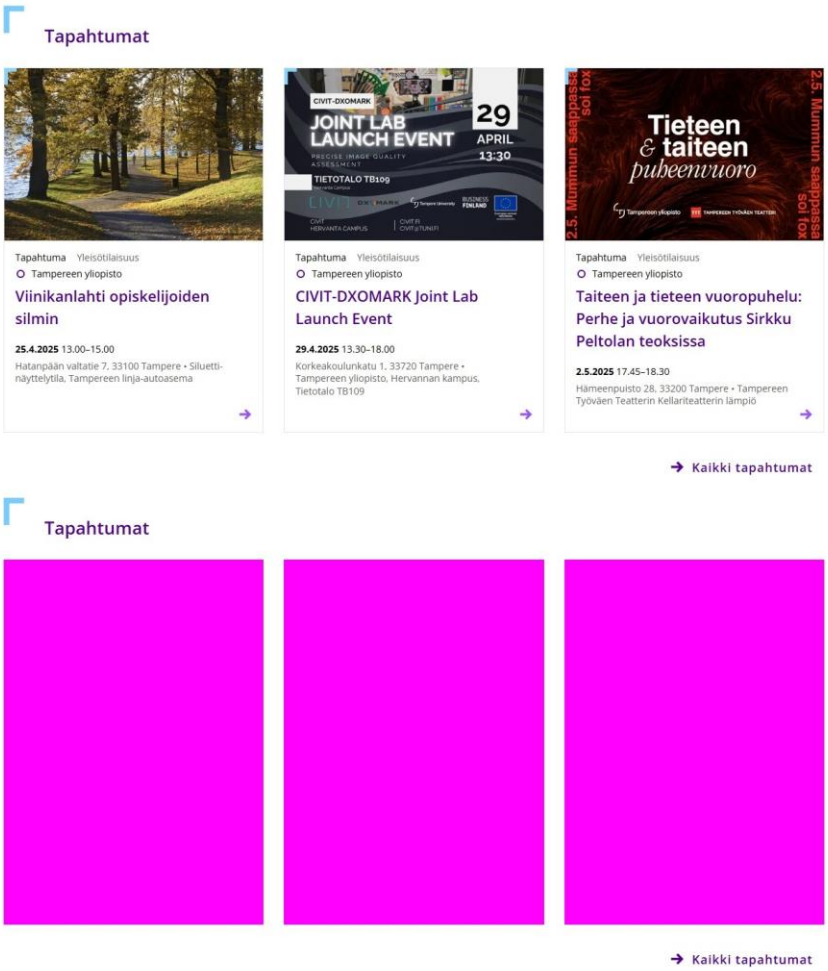
Kuvankaappauksia käytetään visuaaliseen vertailuun, kun taas tilannekuvat voivat myös olla tallennettuja HTML-rakenteita. Kuvankaappauskoodi visuaalisia koko sivun kattavia vertauksia varten näyttää seuraavalta:

```
await expect(page).toHaveScreenshot({ fullPage: true });
```

Sivuilla kuitenkin on usein muuttuvia elementtejä, kuten vaihtuvia uutisotsikoita. Näitä tilanteita varten kuvankaappauksista voi peittää elementit, joita ei halua verrattavaksi. Peittäminen toimii antamalla taulukon paikantimista, joiden valitsemat

elementit halutaan peittää. Kuva 1 näyttää dynaamiset Tapahtuma-elementit peitettynä ja peittämättöminä. Kuvakaappauksissa animaatioiden poisto voi myös vähentää testivirheiden määrää.

```
await expect(page).toHaveScreenshot({
    fullPage: true,
    mask: [page.locator('.dynaaminen-elementti)],
    animations: 'disabled',
});
```



Kuva 1. Maskeeraamaton ja maskeerattu kuvakaappaus.

Kuvakaappausten lisäksi Playwright voi ottaa tilannekuvia HTML-elementeistä ja aria-määritteistä. Kummatkin ovat hyödyllisiä tilanteissa, joissa halutaan tarkastaa suurempi kokonaisuus kirjoittamatta jokaiselle elementille omaa paikanninta

ja tarkastusta. HTML- ja aria-tilannekuvat tallentuvat kuvien sijaan tekstitiedostoihin. Ne otetaan samaan tapaan kuin kuvakaappaukset, mutta kuvien sijaan vertaillaan yksittäisten elementtien tilaa aiempiin tilannekuviin.

```
var content = await page.getByText('Otsikko').textContent();
await expect(content).toMatchSnapshot('test.txt');
```

Kuvankaappaustestien epäonnistuessa ilman hyvää syytä voi ongelman lähde olla esimerkiksi kannettava tietokone, joka ei ole kytketty verkkovirtaan, tai asetustiedostossa määritellyt liian tiukat virhemarginaalit. On siis tärkeää tarkistaa, että testausympäristö on mahdollisimman muuttumaton.

6.5 Palveluun kirjautuminen Playwrightin testeillä

Satoja testejä tehtäessä erityyppisille käyttäjille kirjautuminen voi nopeasti kasvaa hallitsemattomaksi jos jokainen testi kirjautuu sivulle erikseen. Tällöin koodin ylläpidosta tulee vaikeaa ja sen kirjoittamisesta työlästä, testit kestävät tarpeettoman pitkään ja testausympäristön palvelin saattaa estää testiputken kirjautumiset muutaman testin jälkeen.

Yksinkertaisimmillaan kirjautumisen voi saavuttaa lisäämällä ”globalSetup”-tiedoston, joka ajetaan kerran ja jota käytetään tämän jälkeen lähtöpisteenä kaikille testeille. Tämä kuitenkin rajaa kirjautumisen vain yhdelle käyttäjälle, ja tekee olemassaolollaan kirjautumatta testaamisesta lähes yhtä työlään prosessin kuin luoda konfiguraatio, joka tukee kirjautumista useammalla käyttäjällä.

Tilanteissa, joissa haluaa esimerkiksi testata käyttöliittymää sekä normaalilla että järjestelmänvalvojakäyttäjällä, voi konfiguraatiotiedostoon lisätä ensin projektin, joka kirjautuu sivulle ja tallentaa tämän jälkeen evästeet JSON-tiedostoon. Tämän jälkeen voi luoda toisen, ensimmäisestä projektista riippuvaisen projektin, joka käyttää tallennettuja evästeitä.

```
{
    name: 'chromium-setup-admin',
```

```
    use: { ...devices['Desktop Chrome'] },
    testDir: 'setups',
    testMatch: 'setup.admin.spec.ts',
},
{
    name: 'chromium-as-admin',
    use: {
        ...devices['Desktop Chrome'],
        storageState: '.auth/admin-login.json'
    },
    dependencies: ['chromium-setup-admin'],
    testDir: 'admin-tests'
},
```

Esimerkin projekteissa käytettävä ”setup.admin.spec.ts”-tiedosto suorittaa kirjautumisen ja tallentaa evästeet prosessin lopussa tulevia testejä varten. Tämän hoitava tiedosto voi näyttää esimerkiksi seuraavalta:

```
import { test, expect } from '@playwright/test';

test('login setup', async ({ page, context }) => {
    await page.goto('/');
    await page.getByRole(
        'textbox', { name: 'Admin' }
    ).fill('username');
    await page.getByRole(
        'textbox', { name: 'Password' }
    ).fill('password');
    await page.getByRole('button', { name: 'Login' }).click();
    await page.getByRole('heading', { name: 'Logged' }).click();
    await page.context().storageState({ path: '.auth/admin-
login.json' });
    await context.close();
});
```

Muilla testikäyttäjillä kirjautumisessa toistetaan sama prosessi. Projekteihin luodaan kirjautumisen suorittava projekti sekä kirjautumisprojektista riippuva projekti, joka ajaa käyttäjälle tarkoitetut testit.

7 POHDINTA

Opinnäytetyön päätavoitteena oli luoda lähtökohta Playwrightin käyttöönotolle kehitystiimissä, huomioiden sekä työkalun teknisen käyttöönoton että tiimityöhön liittyvät psykologiset näkökulmat. Koen, että tämä tavoite saavutettiin, vaikkakin työssä syvennyttiin vain tietyiltä osin laajaan aiheeseen. Jo ennen työhön paneutumista oletin, että merkittävä este käyttöönotolle olisi olemassa olevat työprosessit sekä erityisesti psykologia ja ryhmädynamiikat. Jälkikäteen tarkasteltuna on olo, että elin lievässä denialismissa näiden inhimillisten tekijöiden todellisesta laajuudesta.

Projektin oikealla suunnalla pitäminen vaati raskasta aiheen rajausta, mutta mielestäni tämä onnistui tyydyttävästi. Työssä olisi ollut helppo takertua ihmiskeskeisiin aiheisiin, jolloin koko Playwright-osuus olisi jäänyt pois. Toinen virhe oli keskittyä pelkkään tekniseen puoleen. Se olisi saattanut kasvattaa itselleni ja lukijalle asiantuntevuuden tunnetta, mutta johtaa sitten työelämässä ongelmiin, jotka johtuvat omista ja tiimin toimintatavoista sekä alitajunnan oikoteistä. Koen, että tasapaino teknisen ja inhimillisen puolen välillä onnistui.

Playwrightiin tutustuessani huomasin, että monet oppimisresurssit eivät nostaneet esille tärkeitä yksityiskohtia sen keskeisimmistä toiminnoista. Virallisen dokumentaation osalta sen laajuus ja jäsennys tekivät siitä epätehokkaan tavan oppia Playwrightin perusteita. Lisäksi oppimisresurssit eivät vaikuttaneet huomioivan testauksen integrointia projektitiimin käytäntöihin. Tämä havainto vahvisti käsitystäni siitä, että dokumentista, joka käsittelee sekä inhimillistä että teknistä puolta, voi olla merkittävää hyötyä Playwrightin käyttöönotossa. Työni pyrkii vastaamaan tähän tarpeeseen tarjoamalla kattavamman näkökulman.

Jatkotutkimukselle ja -kehitykselle näen kolme polkua. Yksi on syvempi paneutuminen tekniikan puoleen, mikä saattaa itsessään haarautua integroitumiseen eri kehitysympäristöihin sekä Playwrightin toimintoihin ja arkkitehtuuriin porautumiseen. Toinen on eri kehitysprosessien vaikutus ohjelmistotestaukseen, varsinkin käyttäjäpolkutestauksessa. Kolmas polku on kaivautuminen syvemmälle psykologian puoleen. Psykologiaan syventymisellä on todennäköisesti suurin potentiaali tehdä vaikuttavia muutoksia kehitystiimeihin työelämässä, mutta se saattaa myös harhautua osittain tietotekniikan alan ulkopuolella oleviin aiheisiin. Playwrightin vertailu muihin testaustyökaluihin tuli mieleen, mutta tuntuu enemmän askeleelta sivuun kuin syvemmälle aiheeseen.

Lopputuloksesta jäi olo, että voin työelämässä tarpeen tullen soveltaa kaikkea työtä tehdessäni oppimaani, ja kysyttäessä osaisin ainakin osoittaa monet kehitystiimit oikeaan suuntaan. En usko, että urani aikaisessa vaiheessa tiettyyn tiimistereotyyppiin tai kehitysympäristöön takertumisesta olisi ollut itselleni suurta hyötyä. Työkaverit ja -paikat vaihtuvat, ja jokainen tiimi on uniikki, joten laajalti sovellettavissa olevat taidot ovat tässä tapauksessa paljon hyödyllisempiä kuin omassa päässäni rakennetun skenaarion ”mestaruus”.

LÄHTEET

Hnaide, S. 2020. We are the SpaceX software team, ask us anything!. Reddit. Haettu 5.6.2025. https://old.reddit.com/r/spacex/comments/gxb7j1/we_are_the_spacex_software_team_ask_us_anything/ft62781/?context=3

Irvine, D. 2023. Svelte with Test-Driven Development

Microsoft. 2025. Playwrightin Github-repositorio. Haettu 5.6.2025. https://github.com/microsoft/playwright/blob/main/packages/playwright-core/src/server/deviceDescriptorsSource.json

Myers, G., Sandler, C. & Badgett T. 2012. The Art of Software Testing. 3. painos. https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf

Playwright. (2025a). .NET Test Runners. Haettu 9.7.2025. https://playwright.dev/dotnet/docs/test-runners

Playwright. (2025b). Java Test Runners. Haettu 9.7.2025. https://playwright.dev/java/docs/test-runners

Playwright. (2025c). Python Test Runners. Haettu 9.7.2025. https://playwright.dev/python/docs/test-runners

Playwright Community. (2022). jest-playwright. Github. Haettu 9.7.2025. https://github.com/playwright-community/jest-playwright

Vitest. (2025). Configuring Playwright. Haettu 9.7.2025. https://vitest.dev/guide/browser/playwright

yyvess. (2023). playwright-mocha. Github. Haettu 9.7.2025. https://github.com/yyvess/playwright-mocha

LIITTEET

Liite 1. esimerkki konfiguraatiotiedostosta

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
    testDir: 'tests',
    testMatch: ['*.spec.ts'],
    testIgnore: ['projectFiles/**'],
    timeout: 40_000, // 30 sekunnin aikakatkaisu.
    use: {
        headless: true, // Ei avaa graafista käyttöliittymää
    },
    reporter: [
        ['list'],
        ['json', {  outputFile: 'results.json' }],
        ['html', { open: 'on-failure' }],
        // ['github'], // Jos projekti käyttää GitHubia
    ],
    projects: [
        {
            name: 'chromium',
            use: { ...devices['Desktop Chrome'] },
        }, {
            name: 'firefox',
            use: { ...devices['Desktop Firefox'] },
        }, {
            name: 'webkit',
            use: { ...devices['Desktop Safari'] },
        },
    ],
    expect: {
        toHaveScreenshot: { threshold: 0.1, maxDiffPixels: 10 },
        toMatchSnapshot: { threshold: 0.1, maxDiffPixels: 10 },
```

```
    },
});
```