VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Joona Luukkonen

WEB-SOVELLUKSEN END-TO-END TESTAUS

CASE Wärtsilä Energy

Liiketalous

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietojenkäsittely

TIIVISTELMÄ

Tekijä	Joona Luukkonen
Opinnäytetyön nimi	Web-Sovelluksen End-To-End Testaus: Case Wärtsilä Energy
Vuosi	2025
Kieli	Suomi
Sivumäärä	38
Ohjaaja	Tero Ulvinen

Opinnäytetyön taustalla on tarve testata Wärtsilän sisäisen Feasibility Tools-alustan web-sovellukseen kuuluvia prosesseja ja ominaisuuksia. Tässä työssä toteutettiin ja suunniteltiin end-to-end testejä sekä asennettiin siihen kuuluva Playwright -niminen testaustyökalu osaksi Feasibility Tools -sovellusta.

Playwright-testaustyökalun valintaan vaikutti sen kyky tukea monipuolisia testausstrategioita sekä sen tarjoama tuki eri selaimille ja laitteille. Työkalu mahdollistaa selainautomaation, mikä mahdollistaa sovelluksen testaamisen käyttäjän näkökulmasta simuloiden oikeita käyttötilanteita. Testitapauksia kehitettiin yhteistyössä sovelluksen kehitystiimin kanssa, jotta kaikki tärkeimmät käyttötilanteet ja mahdolliset ongelmakohdat saatiin huomioitua.

Työn aikana toteutetut end-to-end testit integroitiin osaksi sovelluksen jatkuvan integroinnin ja toimituksen (CI/CD) putkea, mikä mahdollistaa automaattisten testien ajamisen jokaisen koodimuutoksen yhteydessä. Tämä lähestymistapa parantaa merkittävästi sovelluksen laatua ja vähentää manuaalisen testauksen tarvetta.

Avainsanat Playwright, JavaScript, Web-sovellus, ohjelmointi, ohjelmistotestaus

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietojenkäsittely

ABSTRACT

Author	Joona Luukkonen
Title	Web Application End-To-End Testing: Case Wärtsilä Energy
Year	2025
Language	Finnish
Pages	38
Name of Supervisor	Tero Ulvinen

The thesis is based on the need to test the processes and features of Wärtsilä's internal Feasibility Tools web application. In this work, end-to-end tests were implemented and designed, and a related testing tool called Playwright was installed as part of the Feasibility Tools web application.

The choice of the Playwright testing tool was influenced by its ability to support diverse testing strategies, as well as the support it offers for different browsers and devices. The tool enables browser automation, making it possible to test the application from the user's point of view by simulating real use cases. Test cases were developed in cooperation with the application development team to address all key usage scenarios and potential problem areas.

The end-to-end tests implemented during the work were integrated into the continuous integration and delivery (CI/CD) pipeline of the application, enabling automated tests to run with each code change. This approach significantly improves the quality of the application and reduces the need for manual testing.

Keywords Playwright, JavaScript, web application, programming, software testing

SISÄLLYS

KUVAT

KUVIOT

LYHENTEET

REST	Representational state transfer
API	Rajapinta (engl. Application programming interface)
CI/CD	Continuous Integration / Continuous Delivery
BUILD	Ohjelman käännös ja kokoaminen
DEV	Kehitysympäristö (engl. Development environment)
LOCAL	Paikallinen kehitysympäristö (Local environment)
PRODUCTION	Tuotantoympäristö (engl. Production environment)

1 JOHDANTO

Wärtsilä Energyn sisäinen web-sovellus Feasibility Tools Platform yhdistää useimmat energiaan ja voimalaitosmoottoreihin liittyvät mukautetut työkalut, laskelmat ja tietolähteet yhdeksi skaalautuvaksi alustaksi. Feasibility Toolsissa on kaiken kaikkiaan seitsemän erilaista sovellusta ja niiden moduulit on erotettu eri palveluihin, jotka kommunikoivat REST-rajapintojen kautta skaalautuvuuden ja uudelleenkäytettävyyden varmistamiseksi. Tämä mahdollistaa eri prosessien ja niihin liittyvien tietojen saumattoman integroinnin, tuottaen arvokkaita oivalluksia.

Tämä opinnäytetyö on toiminnallinen kehittämistutkimus, jossa suunnitellaan ja toteutetaan web-sovelluksen end-to-end-testiautomaatio. Opinnäytetyö on laadullinen tutkimus, joka on toteutettu projektimuotisena hyödyntäen empiiristä aineistoa, joka perustuu testaukseen, kokeiluihin ja havaintoihin. Analyysissa tarkastellaan testiautomaation kattavuutta, tehokkuutta sekä sen vaikutuksia ohjelmiston laadunvarmistukseen.

End-to-end-testit ovat keskeisiä web-sovellusten laadunvarmistuksessa, sillä ne simuloivat käyttäjän toimintaa ja varmistavat, että kaikki järjestelmän osat toimivat yhteen suunnitellusti. Ne kattavat koko sovelluksen toimintaprosessin aina käyttöliittymästä taustapalveluihin, mikä tekee niistä erittäin kattavia ja tehokkaita virheiden löytämisessä. Tämä on erityisen tärkeää Feasibility Tools -alustan kaltaisessa monimutkaisessa ympäristössä, jossa useat erilliset moduulit ja palvelut toimivat yhdessä.

Tavoitteena on tarjota kattava ja käytännönläheinen ohjeistus end-to-end-testien suunnitteluun ja toteutukseen Feasibility Tools -alustalle. Tarkastellaan ensin mitä on ohjelmistotestaus, jonka jälkeen pohditaan end-to-end-testauksen periaatteita ja sen merkitystä ohjelmistokehityksessä.

Tässä työssä on käytetty ChatGPT:tä ja Microsoft Copilotia ideoinnin, tiedonhaun ja kielentarkistuksen välineenä. Tekoälyä on käytetty tekstin muokkaamisen apuna, että kieli olisi selkeämpää ja helpommin ymmärrettävää, mutta välittäisi asiat yhä alkuperäisen tarkoituksen mukaisesti. Tekoälyä käytettäessä on huolehdittu tekijänoikeuksien kunnioittamisesta ja varmistettu luotettavuus luotettavasta lähteestä.

Tuloksia hyödynnetään Feasibility Tools -alustan laadun ja luotettavuuden parantamiseksi, mikä puolestaan parantaa käyttäjäkokemusta ja vähentää virheiden määrää tuotantoympäristössä. Tavoitteena on osoittaa, kuinka tehokkaasti ja käytännöllisesti modernin web-sovelluksen end-to-end-testaustyökaluilla voidaan varmistaa sovelluksen toimivuus.

2 TUTKIMUKSEN TAUSTA JA TARKOITUS

Feasibility Tools -alustan monimutkaisuus ja useiden eri moduulien sekä palveluiden yhteensovittaminen tuovat mukanaan haasteita sovelluksen testaamisessa. Lähtökohtana Feasibility Tools -alustalla oli niukasti testejä ja todettiin tarpeelliseksi uuden testaustyökalun käyttöönoton.

Sovelluksen tulee toimia saumattomasti ja sen eri osien on kommunikoitava tehokkaasti keskenään. Tämä tekee manuaalisesta testauksesta aikaa vievää ja virheellistä, koska se ei välttämättä kata kaikkia mahdollisia käyttötilanteita ja skenaarioita. Lisäksi manuaalinen testaus voi olla tehotonta jatkuvasti päivittyvässä ja kehittyvässä ympäristössä, jossa koodimuutokset ovat yleisiä.

Tämän tutkimuksen tarkoituksena on kehittää ja toteuttaa automatisoitu end-to-end-testausjärjestelmä Feasibility Tools -alustalle käyttäen Playwright-nimistä testaustyökalua. Tutkimuksen päätavoitteena on parantaa sovelluksen laatua ja luotettavuutta vähentämällä manuaalisen testauksen tarvetta, nopeuttamalla testausprosessia ja vähentämällä virheiden määrää tuotantoympäristössä.

Tämä ei ainoastaan paranna käyttäjäkokemusta, vaan myös säästää kehitystiimin aikaa ja resursseja, jotka muutoin käytettäisiin virheiden korjaamiseen ja manuaaliseen testaukseen. Playwrightin integrointi CI/CD-putkeen nopeuttaa testausprosessia siten, että end-to-end-testit suoritetaan automaattisesti julkaisua tehdessä. Tämä tekee testien suorittamisesta vaivatonta, koska kehittäjät näkevät testien tulokset Bamboossa ilman manuaalisia testausvaiheita.

3 TOTEUTUSMENETELMÄT

Web-sovellusten laadunvarmistuksessa ohjelmistotestaus ja CI/CD-putket ovat keskeisiä tekijöitä, jotka auttavat varmistamaan sovelluksen toimivuuden, laadun ja käyttäjäkokemuksen. Etenkin monimutkaisissa alustoissa, kuten Feasibility Tools Platform, joissa useita erillisiä sovelluksia ja palveluja toimii yhdessä, nämä tekijät mahdollistavat kaikkien sovelluksen osien toimivuuden yhtenäisesti.

Automaatiotestauksen päätavoitteena on parantaa sisäisen web-sovelluksen laatua ja luotettavuutta sekä minimoida manuaalisen testauksen tarvetta, joka voi olla aikaa vievää ja altis virheille. Automaatiotestaus mahdollistaa testien toiston ja laajan kattavuuden, mikä auttaa tunnistamaan potentiaalisia ongelmia nopeasti ja niistä johtuvien virheidenpäätymistä tuotantoon.

CI/CD-putken integrointi puolestaan mahdollistaa jatkuvan testaamisen jokaisen koodimuutoksen yhteydessä, tuoden mukanaan reaaliaikaisen palautteen kehittäjille. Tämä lähestymistapa auttaa nopeuttamaan julkaisuprosessia ja takaa paremman sovelluksen laadun, sillä potentiaaliset virheet ja yhteensopivuusongelmat havaitaan jo kehitysvaiheessa.

3.1 Ohjelmistotestaus

Ohjelmistotestaus on olennainen osa ohjelmistokehitystä ja sen tavoitteena on varmistaa, että ohjelmisto toimii odotetulla tavalla ja täyttää sille asetetut vaatimukset. Testauksen avulla pyritään tunnistamaan virheitä, varmistamaan järjestelmän luotettavuus ja parantamaan ohjelmiston laatua ennen sen käyttöönottoa (Myers, 2012).

3.1.1 Testauksen tavoitteet ja merkitys

Ohjelmistotestauksen päätavoitteisiin kuuluvat virheiden tunnistaminen, ohjelmiston suorituskyvyn ja luotettavuuden varmistaminen, käytettävyyden arviointi sekä vaatimustenmukaisuuden tarkistaminen (Sommerville, 2015).

On tärkeää huomioida, että ohjelmistovirheiden tunnistaminen on keskeinen osa ohjelmistotestausta. Jorgensen (2016) toteaa, että virheiden tunnistaminen on yksi testauksen keskeisimmistä tehtävistä, sillä ohjelmistovirheet voivat johtaa vakaviin seurauksiin, kuten tietoturva-aukkoihin tai järjestelmän toimimattomuuteen. Suorituskyvyn ja luotettavuuden arviointi auttaa varmistamaan, että ohjelmisto toimii vakaasti ja tehokkaasti erilaisissa käyttötilanteissa. Käytettävyyden arvioinnissa tarkastellaan ohjelmiston käyttäjäystävällisyyttä, ja vaatimustenmukaisuuden tarkistamisessa varmistetaan, että ohjelmisto vastaa sille asetettuja liiketoimintatarpeita.

3.1.2 Testauksen menetelmät ja tyypit

Ohjelmistotestaus voidaan jakaa useisiin eri menetelmiin ja tyyppeihin riippuen testauksen tavoitteista ja kohteesta. Manuaalinen testaus edellyttää testaajien suorittavan testit käsin, kun taas automatisoidussa testauksessa käytetään testausohjelmistoja ja -skriptejä testien suorittamiseen automaattisesti. Automatisointi nopeuttaa testausprosessia ja vähentää inhimillisten virheiden mahdollisuutta (Fewster & Graham, 1999).

Funktionaalinen testaus keskittyy ohjelmiston ominaisuuksiin ja niiden toimivuuteen määriteltyjen vaatimusten mukaisesti. Hetzel (1988) korostaa, että ei-funktionaalisessa testauksessa puolestaan tarkastellaan ohjelmiston suorituskykyä, turvallisuutta, skaalautuvuutta ja muita laadullisia ominaisuuksia.

3.1.3 Testauksen eri tasot

Testauksen eri tasot kattavat yksikkötestauksen, integraatiotestauksen ja järjestelmätestauksen. Yksikkötestauksessa keskitytään yksittäisiin moduuleihin tai komponentteihin, jolloin varmistetaan niiden toimivuus. Sommerville (2015) painottaa, että integraatiotestauksessa testataan, kuinka hyvin eri moduulit toimivat yhdessä, kun taas järjestelmätestauksessa arvioidaan ohjelmistoa kokonaisuutena sen varmistamiseksi, että se täyttää kaikki sille asetetut vaatimukset.

End-to-end testaus, johon tässä tutkimuksessa perehdytään, on osa järjestelmätestausta ja sillä on oma erityinen roolinsa. Se simuloi käyttäjän toimintaa koko järjestelmän läpi ja testaa ohjelmiston toimivuutta realistisissa käyttötilanteissa.

Hyväksymistestaus on ohjelmistokehityksen loppuvaiheessa suoritettava testi, jossa varmistetaan, että ohjelmisto vastaa asiakkaan tai loppukäyttäjän tarpeita ennen lopullista käyttöönottoa (Myers, 2012).

3.1.4 Ohjelmistotestauksen haasteet

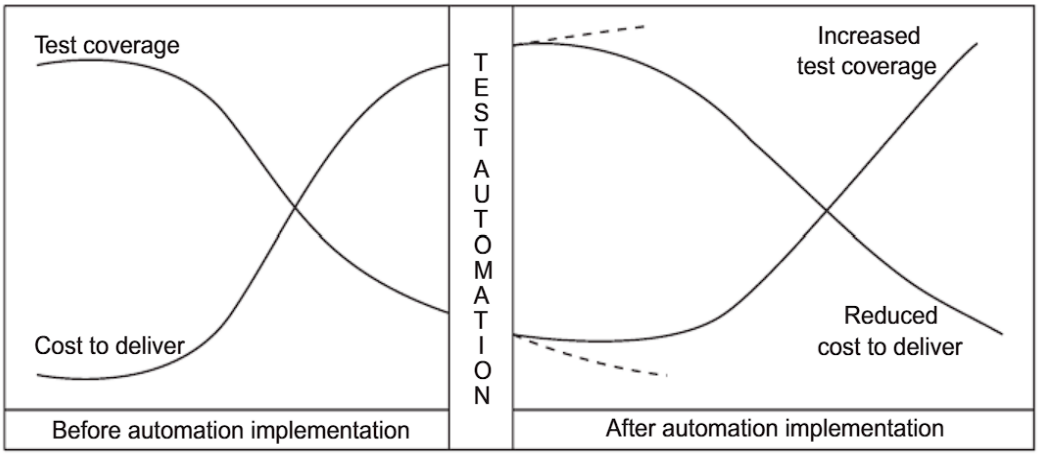
Vaikka ohjelmistotestaus on välttämätön osa ohjelmistokehitystä, sen toteuttaminen ei ole aina ongelmatonta. Testauksen laajuus ja monimutkaisuus voivat aiheuttaa haasteita, erityisesti suurissa ohjelmistoprojekteissa, joissa testattavaa materiaalia on valtavasti. Aikataulu- ja budjettirajoitukset voivat myös vaikuttaa testauksen kattavuuteen, sillä perusteellinen testaus vaatii resursseja ja aikaa. Lisäksi ohjelmistokehityksen aikana tapahtuvat vaatimusten muutokset voivat vaikeuttaa testauksen hallintaa. ja edellyttävät joustavuutta testausstrategiassa.

Automaatiojärjestelmien ylläpito on toinen merkittävä haaste. Automatisoidut testit vaativat jatkuvaa päivitystä, sillä ohjelmistojen muuttuessa myös testiskriptien on pysyttävä ajan tasalla. Tämä voi lisätä testauksen kokonaiskustannuksia ja vaatia erikoisosaamista.

3.2 Automaatiotestaus

Testiautomaatio tarkoittaa testityökalujen käyttöä sovellusten testaamiseen automaattisesti ohjelmistokehityksessä. Samoin kuin ohjelmistokehityksellä yleisesti, testiautomaatiolla on elinkaari, joka määrittää, miten se alkaa, kehittyy ja päättyy. Testiautomaatio kattaa koko ohjelmistotestauksen prosessin ja sisältää erilaisia testaustoimintoja, kuten testitapausten suunnittelun, testiskriptien kirjoittamisen, testien suorittamisen, testien arvioinnin sekä testitulosten raportoinnin (Pyhäjärvi, 2020).

Automaatiotestauksen päätavoitteisiin kuuluu ohjelmiston virheiden tunnistaminen ja laadun varmistaminen kustannustehokkaasti. (Koskela, 2013) toteaa, että automatisoidut testit parantavat tuottavuutta ja mahdollistavat kehitysvauhdin saavuttamisen ja ylläpitämisen.



Kuvio 1. Ennen ja jälkeen testiautomaation käyttöönottoa (Gupta 2024).

Kuviossa 1 esitetään kuinka testiautomaatio voi auttaa kattamaan suuremman osan ohjelmistosta.

Automaatiotestien testiskriptit suoritetaan huomattavasti nopeammin kuin manuaaliset testit. Testiskriptit voidaan suorittaa valvomattomassa tilassa, mikä vähentää testitapausten suorittamisen kustannuksia. Lisäksi testiskriptit eivät ole alttiita inhimillisille virheille (Gupta, 2024).

Testiautomaatiolla saadut tulokset ovat tarkkoja ja luotettavia ja poikkeamat on helppo havaita testiraporttien avulla. Gupta (2024) huomauttaa, että tämä prosessi parantaa ohjelmiston laatua ja varmistaa sen toimivuuden jatkuvasti muuttuvissa olosuhteissa.

3.3 End-to-end (E2E) testaus

E2E-testaus on tärkeä vaihe ohjelmistotestauksessa, jossa testataan koko sovellusjärjestelmän toimintaa alusta loppuun todellisissa tai simuloiduissa käyttöolosuhteissa. Tämä prosessi varmistaa, että kaikki järjestelmän komponentit toimivat saumattomasti yhdessä ja täyttävät

vaaditut toiminnalliset sekä ei-toiminnalliset vaatimukset (Björkman, 2024).

E2E-testiskenaario on joukko vaiheita/toimintoja, joita suoritetaan web-sovelluksessa, kuten käyttäjätunnuksen syöttäminen, salasanan syöttäminen ja kirjautumispainikkeen napsauttaminen. Yhdestä testiskenaariosta voidaan johtaa yksi tai useampi testitapaus määrittelemällä kussakin vaiheessa käytettävät tiedot ja odotetut tulokset (Memon, 2016).

Kunkin testitapauksen suoritus voidaan automatisoida toteuttamalla testiskripti jonkin olemassa olevan lähestymistavan mukaisesti, kuten ohjelmoitava-DOM-pohjainen paikannus. Memon (2016) painottaa että, valinta eri lähestymistapojen välillä riippuu useista tekijöistä. Esimerkiksi web-sovellusten käyttämästä teknologiasta ja mahdollisista käytettävistä web-testauksen työkaluista.

3.3.1 Hyödyt

E2E-testaus tarjoaa monia etuja, jotka parantavat sovelluksen laatua ja käyttäjäkokemusta. Ensinnäkin se luo reaaliaikaista luottamusta. Eli pystytään varmistamaan, että ydintoiminnot, kuten kirjautumiset ja maksutapahtumat, toimivat sujuvasti. Tämä auttaa havaitsemaan ja ratkaisemaan ongelmia ennen kuin ne päätyvät loppukäyttäjien kohdattaviksi (Katalon, 2024).

Nykyajan sovellukset perustuvat usein moniin eri palveluihin ja integraatioihin. (Katalon, 2024) korostaa, että E2E-testaus varmistaa näiden osien yhteensopivuuden ja oikean tiedonkäsittelyn. Tämä on ratkaisevaa, jotta sovelluksen toiminnot eivät kärsi palveluiden tai integraatioiden puutteellisesta yhteensopivuudesta.

E2E-testit tarjoavat myös regressiosuojan. Kun testit ovat käytössä, uusien koodimuutosten aiheuttamien vikojen riski pienenee. Ne toimivat niin kutsuttuna turvaverkkona, joka auttaa pitämään sovelluksen vakaana sen kehittyessä ja muuttuessa (Katalon, 2024).

3.3.2 Haasteet

E2E-testaus tuo mukanaan haasteita, jotka voivat vaikeuttaa sen tehokasta käyttöä. Yksi yleisimmistä ongelmista on testien epävakaus.

Testit voivat epäonnistua satunnaisesti esimerkiksi verkko-ongelmien tai ajoitusongelmien takia, mikä johtaa epäluotettaviin tuloksiin (Katalon, 2024).

Testien pitkä suoritusaika on myös haasteena, koska E2E-testit yleensä kattavat kokonaisia työnkulkuja ja tällöin palautteen saaminen viivästyy. Myös testidatan hallinta voi olla monimutkaista. Testit vaativat tarkkaan määriteltyjä datatiloja, joita voi olla vaikea hallita. Jos sovellus käyttää kolmannen osapuolen rajapintoja, se voi myös aiheuttaa testien epäonnistumista. Katalon (2024) huomauttaa, että E2E-testien ylläpito voi myös vaatia paljon työtä, jos käyttöliittymä tai työnkulku muuttuu usein.

3.4 CI/CD putki

Ohjelmistokehityksen nopeatahtisessa maailmassa korkean laadun ylläpitäminen samalla, kun toimitusnopeutta kiihdytetään, on ensiarvoisen tärkeää. Jatkuvan integraation (Continuous Integration, CI) ja jatkuvan toimituksen (Continuous Deployment, CD) käytännöt ovat nousseet keskeisiksi keinoiksi näiden tavoitteiden saavuttamisessa (Narendar, 2022).

Integroidessaan testiautomaation CI/CD-putkistoihin organisaatiot voivat varmistaa, että ohjelmistoja testataan ja otetaan käyttöön jatkuvasti, mikä Narendarin (2022) mukaan lyhentää markkinoille pääsyä ja parantaa luotettavuutta.

CI/CD-putkistot ovat osa ohjelmistokehityksen käytäntöä, joka automatisoi koodimuutosten integroinnin, testauksen ja käyttöönoton. Sen tavoitteena on nopeuttaa ohjelmistojen toimitusprosessia ja parantaa laatua vähentämällä manuaalisten vaiheiden määrää (Guru99, 2024).

3.4.1 Jatkuva integrointi (CI)

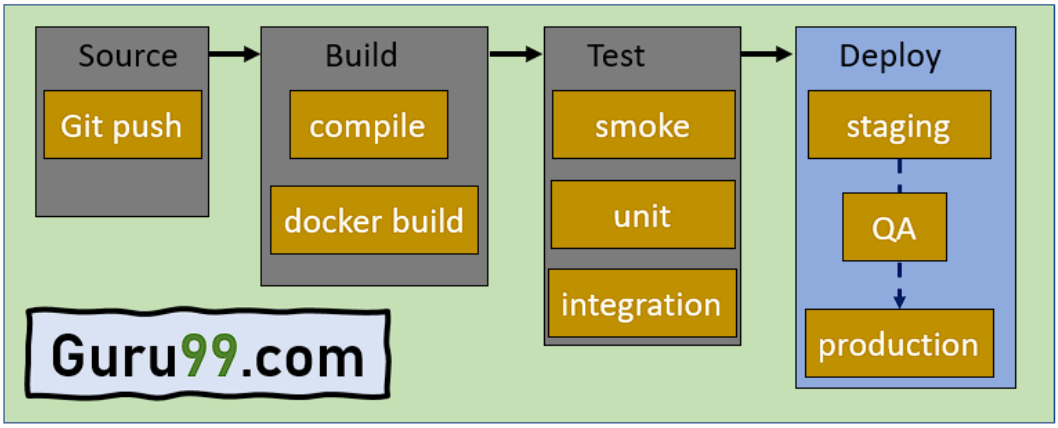
Jatkuva integraatio (CI) on käytäntö, jota kehitystiimit käyttävät koodin automaattiseen yhdistämiseen ja testaamiseen. CI auttaa havaitsemaan virheitä varhaisessa kehitysvaiheessa, mikä tekee niiden korjaamisesta edullisempaa. Automaattiset testit suoritetaan osana CI-prosessia laadun varmistamiseksi. CI-järjestelmät tuottavat artefakteja ja välittävät ne julkaisuprosesseihin, mahdollistamalla tiheät käyttöönotot (Microsoft, 2023).

3.4.2 Jatkuva toimitus (CD)

Jatkuva toimitus (CD) on prosessi, jossa koodi rakennetaan, testataan ja otetaan käyttöön yhdellä tai useammalla testi- ja tuotantoympäristöllä. Useissa ympäristöissä tapahtuva käyttöönotto ja testaaminen parantaa laatua. CD-järjestelmät tuottavat käyttöön otettavia artefakteja, kuten infrastruktuuria ja sovelluksia. Automaattiset julkaisuprosessit hyödyntävät näitä artefakteja uusien versioiden ja olemassa olevien järjestelmien korjausten julkaisemiseksi. Järjestelmät, jotka valvovat ja lähettävät hälytyksiä, toimivat jatkuvasti tarjoten näkyvyyttä koko CD-prosessiin (Microsoft, 2023).

3.5 CI/CD Vaiheet

CI/CD-putki (pipeline) on määritelmä vaiheista, jotka kehittäjän tulee suorittaa toimittaakseen uuden version ohjelmistosta. Jokaisen vaiheen epäonnistuminen laukaisee ilmoituksen esimerkiksi sähköpostilla tai muilla viestintäalustoilla. Tämä mahdollistaa sen, että vastuussa olevat kehittäjät saavat tiedon tärkeistä ongelmista (Guru99, 2024).



Kuvio 2. CI/CD-putken tärkeimmät vaiheet (Guru99 2024).

Kuviossa 2 esitetään tyypillinen CI/CD-putken toimintamalli, jossa vaiheet etenevät järjestyksessä. Tämä toimintamalli auttaa havaitsemaan virheitä varhaisessa vaiheessa ja varmistaa sovelluksen laadukkaan viennin tuotantoon.

Lähde (Source)

Lähdevaiheessa CI/CD-putki käynnistyy koodivarastosta, kun kehittäjä siirtää koodimuutokset versiohallintaan. Kaikki ohjelmaan tehtävät muutokset laukaisevat ilmoituksen CI/CD-työkalulle, joka suorittaa vastaavan putken (Guru99, 2024).

Rakennus (Build)

Rakennusvaihe on CI/CD-putken toinen vaihe, jossa lähdekoodi yhdistetään sen riippuvuuksiin. Tämä tehdään pääasiassa, jotta saadaan rakennettua ohjelmiston ajettava versio, joka voidaan mahdollisesti toimittaa loppukäyttäjälle (Guru99, 2024).

Testaus (Test)

Testausvaiheessa suoritetaan automatisoituja testejä, joilla varmistetaan koodin oikeellisuus ja ohjelmiston toiminta. Tämä vaihe estää helposti toistettavien virheiden päätymisen loppukäyttäjien saataville. Kehittäjien vastuulla on kirjoittaa automatisoidut testit (Guru99, 2024).

Julkaisu (Deploy)

Tämä on viimeinen vaihe, jossa tuote julkaistaan. Kun rakennusvaihe on onnistuneesti läpäissyt kaikki tarvittavat testiskenaariot, tuote on valmis otettavaksi käyttöön tuotantopalvelimella (Guru99, 2024).

4 TYÖKALUT

Tässä luvussa käsitellään Feasibility Tools -alustan jatkokehityksessä käytettyjä työkaluja, jotka tukevat sovelluksen kehitystä ja testausta. Työkalut valittiin huolellisesti niiden tarjoamien ominaisuuksien ja yhteensopivuuden perusteella ja niitä hyödynnettiin sekä ohjelmoinnissa että testauksen automatisoinnissa.

4.1 Javascript

Javascript on Reactin ohjelmointikieli ja yksi keskeisistä työkaluista Feasibility Tools -alustan kehityksessä. Javascript mahdollistaa dynaamisten ja interaktiivisten web-sovellusten luomisen ja se tarjoaa laajan valikoiman kirjastoja ja kehitystyökaluja, jotka nopeuttavat ohjelmistokehitystä. Javascriptin käyttämisen etuina ovat sen selkeys ja laaja ekosysteemi, joka mahdollistaa nopean ongelmanratkaisun kehitysprosessissa (Javascript, 2024).

4.2 React

React on Meta:n kehittämä JavaScript-pohjainen kirjasto, joka mahdollistaa modulaaristen ja tehokkaiden käyttöliittymien rakentamisen (React, 2024).

Feasibility Tools -alustan käyttöliittymä on toteutettu Reactilla, mikä tekee sovelluksen koodista selkeää ja helposti ylläpidettävää. React hyödyntää komponenttipohjaista arkkitehtuuria, joka tukee uudelleenkäytettävien koodikomponenttien luomista. React myös skaalautuu hyvin, minkä ansiosta se soveltuu suurten ja monimutkaisten web-sovellusten kehitykseen (React, 2024).

4.3 Playwright

Playwright on Microsoftin kehittämä moderni selainautomaatiotyökalu, jota käytettiin Feasibility Tools -alustan end-to-end-testien toteutuksessa.

Playwrightin valinnan perusteena olivat sen monipuolisuus, tuki eri selaimille ja laitteille sekä kyky simuloida käyttäjän vuorovaikutusta sovelluksen kanssa. Playwright tarjoaa kehittyneet työkalut testiskripteille ja helpottaa monimutkaisten testitapausten toteuttamista (Microsoft, 2024).

Vaikka Playwright oli luonnollinen valinta Feasibility Tools -alustan end-to-end-testeihin sen monipuolisuuden ja kehittyneiden ominaisuuksien vuoksi, markkinoilla on myös muita tehokkaita e2e-testaustyökaluja.

Esimerkiksi Selenium on suosittu avoimen lähdekoodin työkalu, joka tukee monia ohjelmointikieliä ja selaimia. Cypress on toinen moderni vaihtoehto, joka on erityisen käyttäjäystävällinen ja helppokäyttöinen, vaikka sen selainkäyttö on rajoitetumpi. Nämä vaihtoehdot tarjoavat erilaisia etuja ja ominaisuuksia, jotka voivat vastata eri projektien tarpeisiin.

4.4 Bamboo

Bamboo on Atlassianin kehittämä CI/CD-työkalu, jota hyödynnettiin automatisoitujen testien suorittamisessa ja jatkuvassa integroinnissa. Wärtsilässä käytämme myös muita Atlassianin kehittämiä työkaluja, kuten Bitbucketia, Jiraa ja Confluencea.

Bamboo mahdollistaa saumattoman integroinnin muiden kehitystyökalujen, kuten Gitin ja Dockerin kanssa ja sen avulla voidaan hallita sovelluksen julkaisuprosessia. Bamboo auttaa kehittäjiä varmistamaan, että jokainen koodimuutos on testattu perusteellisesti ennen tuotantoon siirtämistä (Atlassian, 2024).

4.5 Docker

Docker on konttiteknologiaan perustuva työkalu, joka mahdollistaa sovellusten ja niiden riippuvuuksien paketoimisen eristettyihin ympäristöihin. Dockerin avulla voidaan kehittää ja testata sovelluksia yhdenmukaisessa ympäristössä riippumatta kehittäjän tai palvelimen käyttöjärjestelmästä. Tämä vähentää ympäristöjen eroista johtuvia virheitä ja helpottaa sovelluksen jakelua (Docker, 2024).

5 TOTEUTUS

Tässä luvussa käymme läpi, kuinka Playwright-työkalu asennetaan ja konfiguroidaan osaksi FeasibilityTools-työkalua. Lisäksi toteutamme testejä ja tutustumme Playwrightin ominaisuuksiin ja lisäosiin. Luvun lopuksi käsittelemme, miten testitapaukset integroidaan osaksi olemassa olevaa CI/CD-putkea.

5.1 Playwright työkalun asennus

Playwright on Microsoftin kehittämä E2E-testaustyökalu. Valitsimme Playwrightin, koska se tukee monia käyttöjärjestelmiä ja selaimia. Lisäksi se on suhteellisen helppo käyttää ja tukee headless-tilaa, mikä helpottaa testien integrointia CI/CD-putkeen. Playwright tukee myös useita ohjelmointikieliä ja internetistä löytyy laaja dokumentaatio työkalun käyttöön.

Playwrightin asentamiseksi siirrymme komentokehotteella client-kansioon, jossa React-projekti sijaitsee.

Aja seuraava komento client-kansion juuressa: npm init playwright@latest

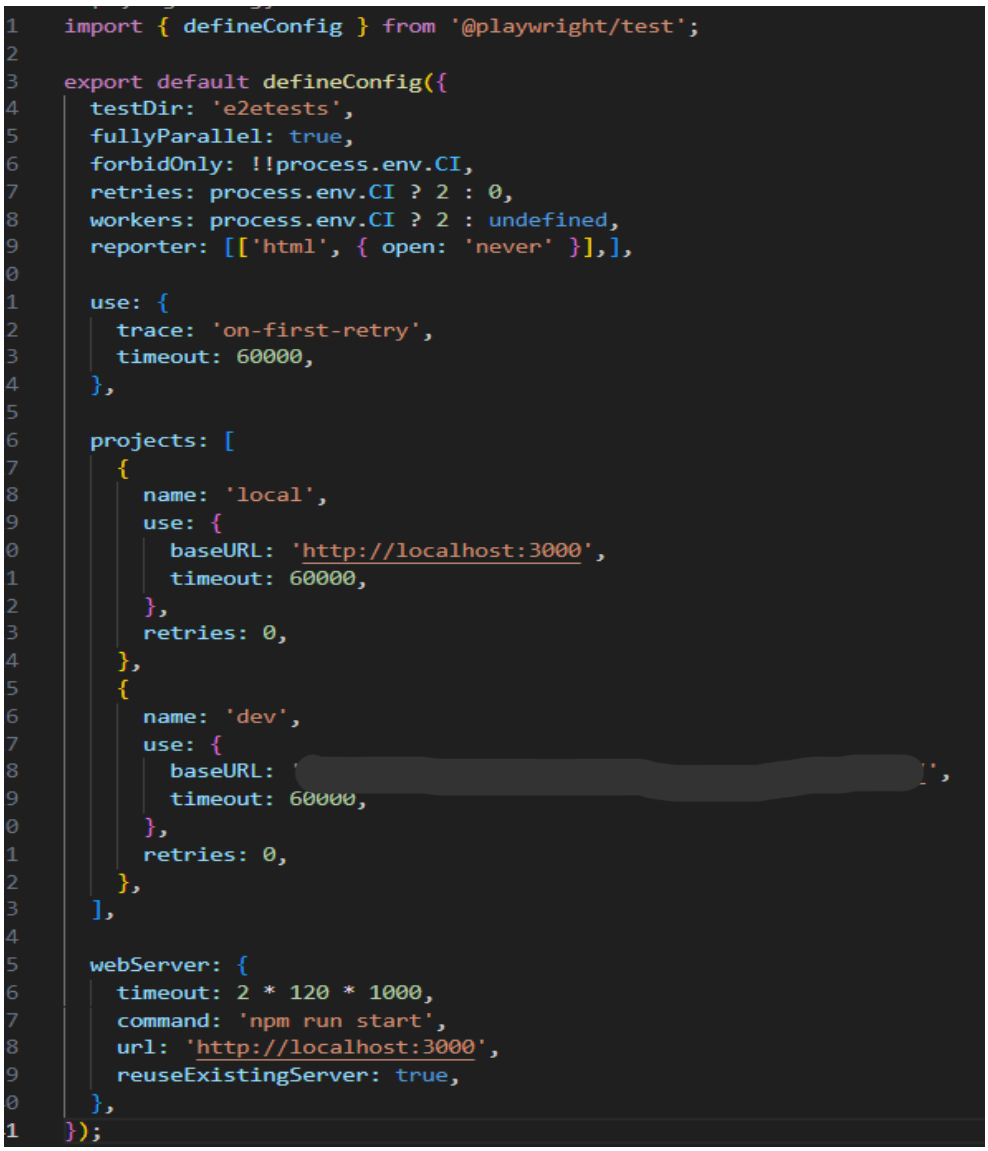
Asennuksen yhteydessä kysytään, asennetaanko Playwright JavaScriptillä vai TypeScriptillä. TypeScript on valittu oletuksena, mutta koska meidän projektissamme käytämme JavaScriptiä, valitsemme JavaScriptin.

Lisäksi asennuksen aikana pyydetään nimeämään testikansio, johon kaikki E2E-testit tallennetaan. GitHub Actions -integraation voi myös lisätä halutessaan, mutta käytämme Bitbucket-nimistä ohjelmaa.

Lopuksi asennuksessa pyydetään valitsemaan selaimet, joissa testit ajetaan. Oletuksena on valittu kaikki selaimet ja pidämme tämän valinnan.

5.2 Konfigurointi

Muokataan playwright.config.js tiedostoa, jotta testit tulevat toimimaan lokaalissa ympäristössä ja CI/CD putkessa.



```
import { defineConfig } from '@playwright/test';

export default defineConfig({
  testDir: 'e2etests',
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? 2 : undefined,
  reporter: [['html', { open: 'never' }],],

  use: {
    trace: 'on-first-retry',
    timeout: 60000,
  },

  projects: [
    {
      name: 'local',
      use: {
        baseURL: 'http://localhost:3000',
        timeout: 60000,
      },
      retries: 0,
    },
    {
      name: 'dev',
      use: {
        baseURL: '[redacted]',
        timeout: 60000,
      },
      retries: 0,
    },
  ],

  webServer: {
    timeout: 2 * 120 * 1000,
    command: 'npm run start',
    url: 'http://localhost:3000',
    reuseExistingServer: true,
  },
});
```

Kuva 1. Playwright.config.js tiedosto

TestDir määrittää, missä testit sijaitsevat. FullyParallel mahdollistaa testien ajamisen rinnakkain, mikä parantaa testausprosessin tehokkuutta. ForbidOnly estää test.only -testien ajamisen CI-ympäristössä, jolloin vältetään virheelliset suoritukset, joissa vain yksi testi ajetaan. Retries määrittää, kuinka monta kertaa testejä yritetään ajaa CI-ympäristössä

epäonnistumisen jälkeen. Workers määrittää, kuinka monta työtekijää (workeria) käytetään testien ajamiseen CI-ympäristössä ja tässä tapauksessa niitä on kaksi. Lopuksi Reporter-kohdassa raportointityökalu ei käynnisty automaattisesti.

Use-kohdassa määritetään yleiset asetukset, kuten trace-jäljitys ja timeout-aika, jotka vaikuttavat testien suoritukseen ja virheiden jäljittämiseen.

Projects-kohdassa määritellään eri projektit ja niiden vastaavat asetukset. Ensimmäinen projekti on local, jonka osoite on localhost:3000 – tätä käytetään, kun luodaan ja ajetaan uusia testejä paikallisessa ympäristössä. Toinen projekti on dev, joka on määritelty käytettäväksi CI/CD-putkissa, eli testejä ajetaan tässä ympäristössä kehitysvaiheessa.

Webserver-kohdassa määritetään web-palvelimen asetukset, jotka vaikuttavat siihen, kuinka palvelin käynnistetään ja kuinka se käyttäytyy testien aikana. Timeout-asetus määrittää aikarajan, jonka aikana palvelimen tulee käynnistyä ennen kuin testausprosessi jatkuu. Command-kohdassa määritellään komento, jolla palvelin käynnistetään ja url-kohdassa määritellään palvelimen URL-osoite, jossa palvelin on saatavilla testien aikana.

ReuseExistingServer-asetus mahdollistaa olemassa olevan palvelimen uudelleenkäytön, mikäli se on mahdollista, jolloin palvelinta ei tarvitse käynnistää jokaista testiä varten uudelleen.

5.3 Testitapauksen toteutus ja suorittaminen

Playwrightin testit voidaan luoda joko kirjoittamalla ne itse tai nauhoittamalla. Nauhoittaminen on nopea ja vaivaton tapa luoda testitapaus, erityisesti jos halutaan nopeasti luoda yksinkertaisia testejä ilman ma-

nuaalista koodin kirjoittamista. Nauhoittaessa käyttäjä voi toimia verkkosivulla normaalisti ja Playwright generoi koodirivit automaattisesti oikeassa järjestyksessä.

On kuitenkin tärkeää huomioida, että nauhoittamisen aikana sivun elementit voivat joskus muuttua tai olla huonosti tunnistettavissa, jolloin Playwright ei välttämättä löydä oikeaa elementtiä, kuten nappia. Tällöin tulee määrittää elementille yksilöllinen tunniste, kuten id tai muu optimoitu tunnistustapa, joka takaa, että elementti löytyy oikein myös automaattisesti nauhoitetuissa testeissä.

Komennolla npx playwright codegen http://localhost:3000/ voidaan aloittaa testin nauhoittaminen. Tässä esimerkissä käytetään paikallista osoitetta, mutta voidaan käyttää mitä tahansa URL-osoitetta, johon halutaan luoda testin. Tämä komento avaa Playwrightin koodin nauhoitustilan, jossa voi vuorovaikuttaa verkkosivun kanssa normaalisti ja Playwright generoi koodin automaattisesti taustalla.

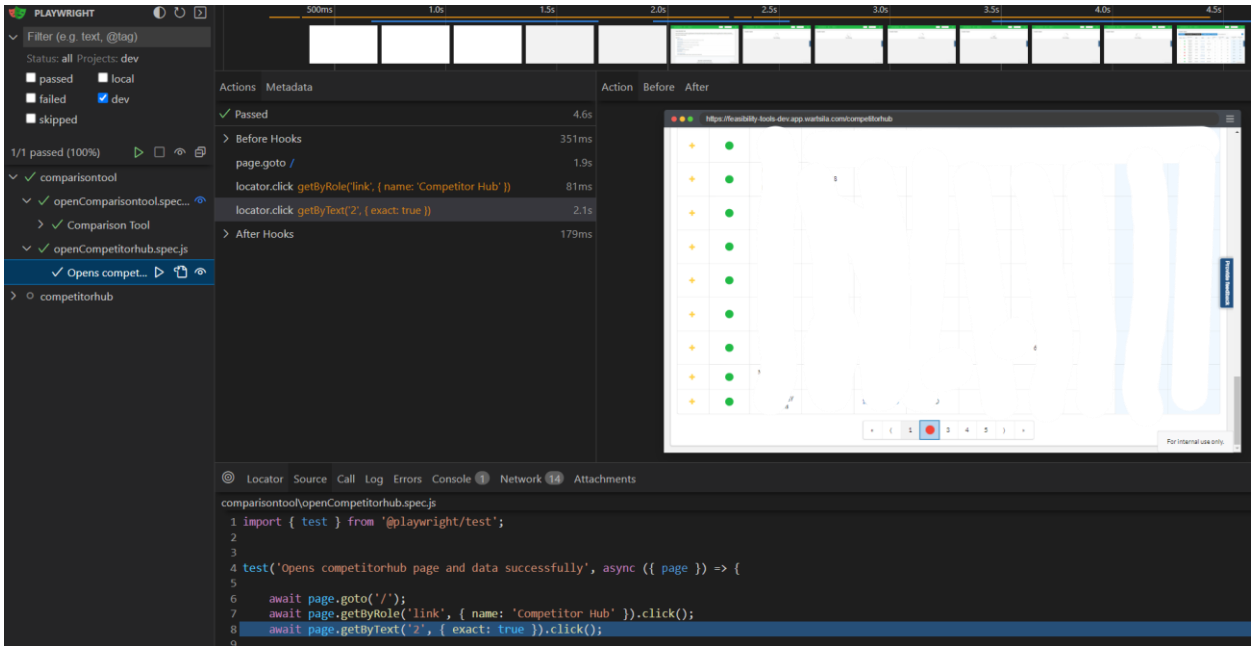
Komennolla npx playwright test voidaan suorittaa kaikki testit. Jos haluat ajaa vain tietyn testin, voi lisätä testitiedoston nimen komennon loppuun, esimerkiksi npx playwright test test-file.spec.js.

Komennolla npx playwright show-report avataan testiraportti, josta voi tarkastella testien suoritustuloksia. Raportista näkee, mitkä testitapaukset ovat onnistuneet ja mitkä epäonnistuneet.

Komennolla npx playwright test –ui voidaan ajaa testejä interaktiivisessa käyttöliittymässä. Tämä avaa Playwrightin UI:n, jossa voi tarkastella testejä ja suorittaa ne visuaalisesti, mikä tekee testien seuraamisesta ja virheiden etsimisestä helpompaa.

5.3.1 Playwright UI näkymän edut

UI-näkymän avulla voidaan tarkastella testitapauksia, suorittaa yksittäisiä testejä, tarkastella testituloksia ja virheraportteja sekä toistaa testejä. Testitulosten tarkastelutoiminnossa käyttäjälle avautuu näkymä, josta voidaan tarkastella testien tuloksia ja nähdä onnistuneet ja epäonnistuneet testit. Lisäksi UI-näkymä mahdollistaa yksittäisten testien suorittamisen, mikä helpottaa testausprosessin hallintaa ja virheiden paikantamista. Virheraporttien tarkastelu tarjoaa yksityiskohtaista tietoa epäonnistuneista testeistä, mikä auttaa kehittäjiä korjaamaan ongelmat nopeasti. Testien toistaminen on myös mahdollista, jolloin voidaan varmistaa, että korjatut virheet eivät toistu.

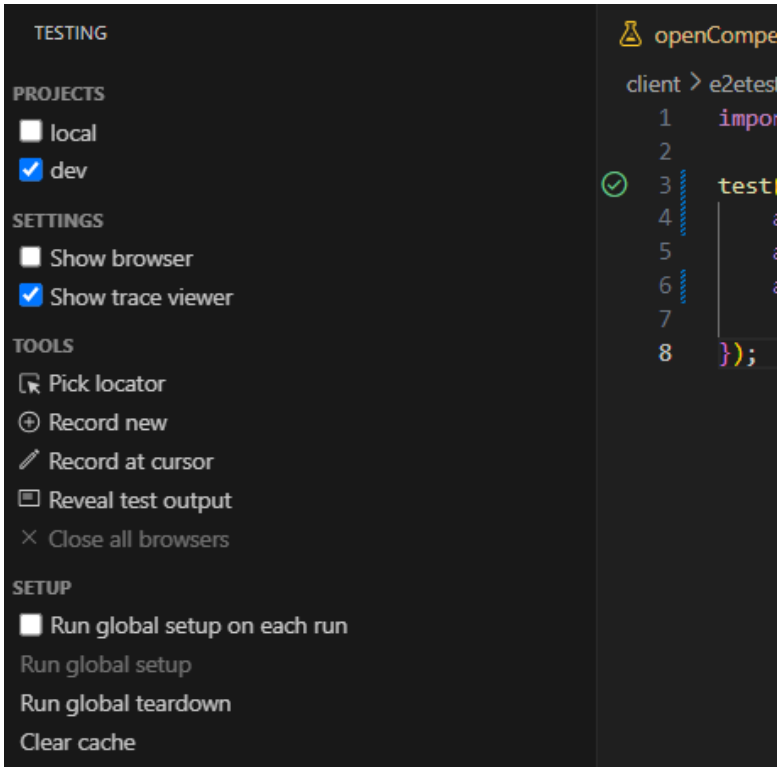


Kuva 2. Playwright UI näkymä

Kuvassa 2 on Playwrightin UI-näkymä, jossa näkyvät testitapaukset, niiden tulokset sekä mahdollisuus suorittaa testejä ja tarkastella virheraportteja. UI-näkymässä näkyy myös alhaalla testin etenemisvaihe, mikä helpottaa testin seuraamista. Rivejä voidaan klikata, jolloin oikealla oleva ikkuna siirtyy kyseiseen kohtaan.

5.3.2 Playwright Test Extension

Visual Studio Codessa Playwright Test extension (lisäosa) tarjoaa kaikki tarvittavat ominaisuudet E2E-testaukseen, kuten testien nauhoittamisen. Tämä lisäosa poistaa tarpeen kirjoittaa Playwright-komentoja käsin ja tarjoaa käytännössä samat toiminnot kuin Playwrightin UI-näkymä, lukuun ottamatta testin nauhoitusominaisuutta.



Kuva 3. Playwright Test VScode lisäosa

Kuvassa 3 näkyy Playwright extension (lisäosa). Lisäosassa voidaan valita helposti ympäristö, millä testit ajetaan.

Lisäksi Tools-kohdasta löytyvät tärkeimmät ominaisuudet, kuten uuden testin nauhoittaminen. Erityisesti "Record at cursor" -ominaisuus on hyödyllinen, sillä sen avulla voi jatkaa testin nauhoittamista tai muokata testiä haluamastaan kohdasta.

Lisäosa tarjoaa myös testin ajamisen helposti. Oikealla näkyvä vihreä hyväksymismerkki osoittaa, että testitapaus on onnistunut. Klikkaamalla merkkiä voi helposti ajaa testin uudelleen.

5.3.3 Testin toteutus

Testitapaus aloitetaan luomalla testitiedosto testikansioon, jonka Playwright luo client-kansion sisään. Tiedosto nimetään muodossa testi.spec.js. Tapauksessani tiedosto on nimetty openCompetitorhub.spec.js.

Playwright Test -lisäosassa on ominaisuus nimeltä "Record new". Klikkaamalla sitä lisätään testikansioon uusi testitiedosto, jossa on valmiiksi pohja ja nauhoitus käynnistyy automaattisesti, mikä nopeuttaa testien luomista.

Kuva 4. Playwright testitapaus

Kuvassa 4 näkyy nauhoitettu yksinkertainen testitapaus. Playwright-testeissä tuodaan test-funktio testikirjastosta, jonka jälkeen määritetään testitapaus.

Rivillä 3 määritellään, mitä testin pitäisi tehdä: testi avaa sivun ja tarkistaa, että data tulee näkyviin. Kun testifunktio on määritetty, voidaan aloittaa testin kirjoittaminen tai nauhoittaminen.

Rivillä 4 testi navigoi Feasibility Toolsin etusivulle. Playwright käyttää oletus-URL-osoitetta, joka on määritetty playwright.config.js -tiedostossa. Tämän vuoksi testissä on käytetty merkintää '/'.

Rivillä 5 klikataan Competitor Hub linkkiä.

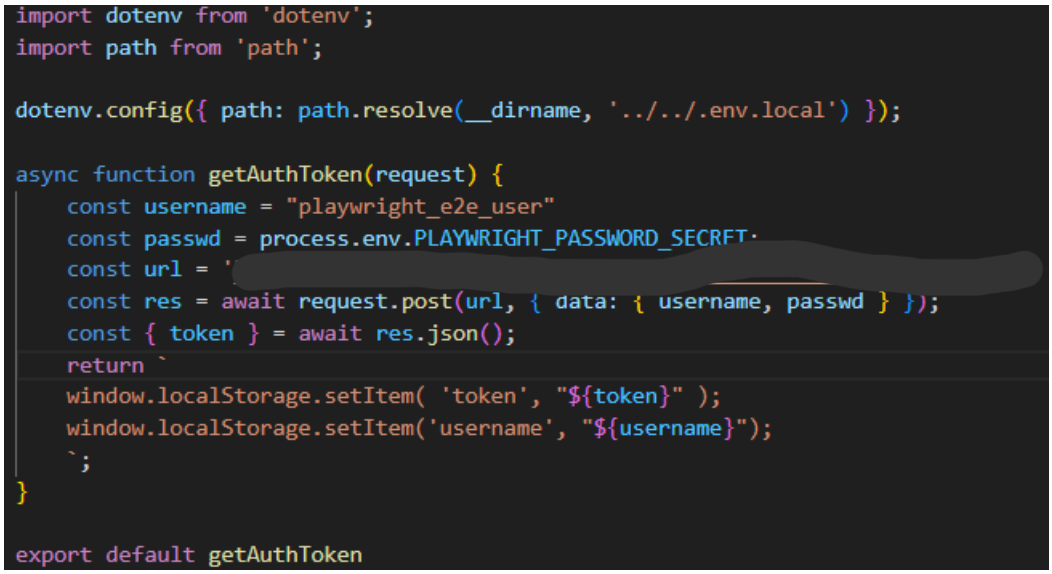
Viimeisenä testissä siirryttään Competitor Hub taulun sivulle 2.

5.4 Testitapaus autentikoinnin kanssa

Feasibility Tools -alusta sisältää seitsemän erilaista työkalua, joista kaksi vaatii kirjautumisen sovellukseen. Suurin osa työkaluista vaatii myös kirjautumisen, jos käyttäjä haluaa käyttää kaikkia niiden ominaisuuksia. Tämä tarkoittaa, että lähes jokainen testitapaus tulee suorittaa kirjautuneena.

Toteutimme kirjautumisen lisäämällä tietokantaan testikäyttäjän nimeltä "playwright_e2e_user". Työkollegani vastasi tämän osuuden toteutuksesta ja kirjoitti tarvittavan koodin Django-backendiin. Backendissä suoritetaan kysely, joka hakee kirjautumismerkin Active Directorylta (AD). Tätä kirjautumismerkkiä tarvitaan käyttäjän autentikointiin.

Frontendissä kirjoitin uuden tiedoston nimeltä PlaywrightUtils.js utils-kansioon. Tiedostossa toteutetaan testikäyttäjän kirjautuminen sekä kirjautumismerkin hakeminen.



```
import dotenv from 'dotenv';
import path from 'path';

dotenv.config({ path: path.resolve(__dirname, '../../.env.local') });

async function getAuthToken(request) {
    const username = "playwright_e2e_user"
    const passwd = process.env.PLAYWRIGHT_PASSWORD_SECRET
    const url = '
    const res = await request.post(url, { data: { username, passwd } });
    const { token } = await res.json();
    return `
    window.localStorage.setItem( 'token', "${token}" );
    window.localStorage.setItem('username', "${username}");
    `;
}

export default getAuthToken
```

Kuva 5. PlaywrightUtils.js tiedosto

Kuvassa 5 näkyy tiedosto, jossa on toteutettu getAuthToken-niminen funktio. Ensin ladattiin npm-komennolla dotenv- ja path-kirjastot ja ne tuodaan tähän tiedostoon.

Dotenv-kirjasto tuodaan ympäristömuuttujien lataamiseksi .env-tiedostosta ja path-kirjasto puolestaan tiedostopolkujen käsittelyä varten. Seuraavaksi ladataan ympäristömuuttujat .env.local-tiedostosta.

Funktiossa getAuthToken määritellään käyttäjänimi, haetaan salasana ja autentikointipalvelun osoite.

Parametreina välitetty request lähettää POST-pyynnön autentikointipalveluun käyttäjätunnuksella ja salasanalla.

Lopuksi res-muuttujasta parsitaan token ja palautetaan koodi, joka tallentaa tokenin ja käyttäjänimen localStorageen.

```
import { test } from '@playwright/test';
import getAuthToken from '../../src/utils/playwrightUtils';

test.describe('Comparison Tool', () => {
    test.beforeEach(async ({ page, context, request }) => {
        await context.addInitScript(await getAuthToken(request));
        await page.goto('/comparison');

    });
    test('navigate to and interact with Comparison Tool', async ({ page }) => {
        await page.getByText('Heavy Fuel Oil').click();
    });
});
```

Kuva 6. Testitapaus autentikoinnin kanssa

Kuvassa 6 näkyy yksinkertainen testi, joka hyödyntää edellä mainittua getAuthToken-funktiota. Ensin haetaan getAuthToken utilsista ja testi-funktio Playwright-kirjastosta.

Tämän jälkeen luodaan test.describe-funktion sisään test.beforeEach. BeforeEach-funktio suorittaa kirjautumisen 'playwright_e2e_user'-käyttäjänimellä ja siirtyy comparisontoolin sivulle joka kerta, kun testi ajetaan näiden sisällä. Tällä hetkellä tiedostossa on vain yksi testitapaus, joten kirjautuminen tapahtuu vain kerran.

5.5 Testien integrointi Bamboon CI/CD putkeen

Feasibility Tools -alusta käyttää Bamboo-nimistä työkalua CI/CD-prosessien kehittämiseen ja ylläpitämiseen. E2E-testien integrointi CI/CD-putkeen tuo mukanaan monia hyötyjä, kuten sen, että se varmistaa testien ajamisen ennen kuin koodimuutokset etenevät eteenpäin ja tekee testien seuraamisesta helppoa.

Bamboo-työkalua konfiguroidaan bamboo.yaml-nimisen tiedoston avulla. Tämä tiedosto määrittää rakennus- ja käyttöönottoprosessit, jotka Bamboo suorittaa automaattisesti.

Tällaisissa tiedostoissa oletusvaiheena on Build. Build-vaihe automatisoi projektin versionhallinnan, testauksen, rakennuksen ja Docker-kuvien

käsittelyn. Tämä vaihe varmistaa, että projekti on valmis käyttöönottoon ja että kaikki tarvittavat artefaktit ovat saatavilla. Artefaktit ovat tiedostoja tai kansioita, jotka pakataan build-vaihetta ajaessa.

```yaml
artifacts:
  - name: build.properties
    pattern: build.properties
    shared: true
    required: true
  - name: helm
    pattern: devops/*
    shared: true
    required: true
  - name: Makefile
    pattern: Makefile
    shared: true
    required: true
  - name: e2etests
    pattern: client/**/*
    shared: true
```

Kuva 7. Bamboo.yaml tiedoston artefaktit

Kuvassa 7 esitetään Feasibilitytoolsin build-prosessissa tarvittavat artefaktit.

Artefaktit ovat:

- Build.properties-tiedosto
- Helm-kaavat
- Makefile-tiedosto
- E2etests-hakemisto

Build.properties-tiedosto sisältää version tiedot ja toimii siten buildin hallintaan liittyvän versionhallinnan osana. Helm on hakemisto, joka sisältää Helm-kaavat (Helm charts). Kaavoilla määritellään sovelluksen asennukseen ja hallintaan liittyvät resurssit, kuten konfiguraatiot, riippuvuudet ja päivityssäännöt. Makefile-tiedosto puolestaan sisältää rakennuskomentoja, joita käytetään muun muassa yksikkötestien suorittamiseen. Lisäksi E2etests-hakemisto sisältää end-to-end-testit, jotka

sijaitsevat client-kansiossa ja mahdollistavat järjestelmän kokonaisvaltaisen testauksen.

5.5.1 Bamboo ympäristöt

Bamboo.yaml-tiedostoon on määritetty ympäristöt, joissa käyttöönotto tapahtuu. Nämä ympäristöt ovat: Development, QA, Production ja Test. Nämä ympäristöt vaativat Docker-imagen toimiakseen.

Docker-kuvan merkitys ympäristökohtaisissa tehtävissä ja muuttujissa on tarjota yhtenäinen ja eristetty ympäristö, jossa tehtävät suoritetaan. Tämä varmistaa, että kaikki riippuvuudet ja työkalut ovat saatavilla ja että ympäristö on riippumaton siitä, missä se suoritetaan.

Development-, QA- ja Production-ympäristöt ovat kaikki samanlaisia ja käyttävät samoja tehtäviä ja Docker-kuvia keskenään. Ainoa ero näissä on URL-osoite.

5.5.2 Test ympäristö

E2E-testit vaativat oman ympäristön, joten loimme Test-nimisen ympäristön. Tämä ympäristö on tarpeen, koska koodimuutokset on ensin julkaistava Development-ympäristöön ennen testien suorittamista.

```
Test:
  triggers:
    - environment-success: Development
  tasks:
    - artifact-download
    - inject-variables: build.properties
    - script:
        - set -e
        - cd client
        - rm -rf node_modules # Remove node_modules to avoid conflicts with the docker container
        - npm install dotenv
        - npm install path
        - npm install @playwright/test
        - npx playwright install
        - export PLAYWRIGHT_PASSWORD_SECRET=${bamboo.e2e_test_user_password}
        - npx playwright test --project=dev
  docker:
    image: mcr.microsoft.com/playwright:v1.44.1-jammy
```

Kuva 8. Test ympäristön projekti

Kuvan 8 projekti sijaitsee bamboo.yaml tiedostossa.

Test-ympäristön tehtävät käynnistetään automaattisesti, kun Development-ympäristön käyttöönotto onnistuu. Tämän jälkeen buildissa luodut artefaktit ladataan ja haetaan.

Muuttujat injektoidaan build.properties-tiedostosta, jotta niitä voidaan käyttää myöhemmissä vaiheissa. Suoritetaan skripti, joka ajaa määritetyt komennot. Komennoilla asennetaan Playwright-testikirjasto ja siihen tarvittavat riippuvuudet.

Lopuksi määritetään Docker-kuva, joka löytyy Playwrightin dokumentaatiosta. Docker-kuvaa käytetään ympäristön tehtävien suorittamiseen.

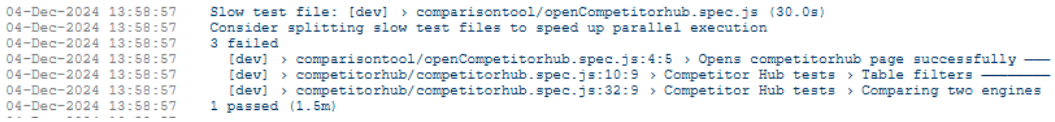
5.5.3 Integroinnin yhteenveto

Kuviossa 3 näkyy tekemäni yksinkertainen kaavio, jossa käsitellään vaiheittain CI/CD integroinnin vaiheet.

Kuvio 3. Kaavio CI/CD vaiheista

1. Kehittäjä vie koodimuutokset Git-versionhallinnan avulla Bitbucketiin.
2. Build-vaihe käynnistyy automaattisesti, kun muutokset on viety Bitbuckettiin.
3. Build-vaihe kestää muutaman minuutin, jonka jälkeen build joko onnistuu tai epäonnistuu.
4. Buildin onnistuessa muutokset voidaan julkaista ja uudet muutokset tulevat näkyviin kuvan 3 Dev-osoitteessa.Test-ympäristö alkaa automaattisesti, kun julkaisu on suoritettu.

Test-ympäristö tulee suorittamaan kaikki kirjoitetut Playwright- testit ja antaa palautteen kehittäjälle onnistuneista ja epäonnistuneista testeistä.



Kuva 9. Test-ympäristön palaute E2E-testeistä

Tulokset näyttivät, että 3 E2E-testeistä epäonnistui ja kuvassa 9 näkyvät Bamboo:n logit näyttävät testitiedoston nimen ja rivivälin missä kohtaa testi on epäonnistunut.

6 JOHTOPÄÄTÖKSET

Opinnäytetyön tavoitteena oli suunnitella, toteuttaa ja asentaa Wärtsilän sisäiseen FeasibilityTools-työkaluun uusi Playwright-niminen testaustyökalu. Tämä työkalu tarjoaa kehittäjille uuden tavan toteuttaa testitapauksia ja parantaa sovelluksen testausprosessia.

6.1 Keskeiset löydökset

E2E-testaustyökalujen avulla kehittäjät voivat automatisoida selaintestauksia eri alustoilla ja selaimilla, mikä lisää testauksen kattavuutta ja luotettavuutta. Tämä on merkittävä parannus verrattuna aiempiin manuaalisiin testausmenetelmiin ja antoi merkittävän lisän olemassa olevien yksikkötestien rinnalle.

Lisäksi havaittiin, että Playwright-testaustyökalun yksityiskohtaiset virheraportit auttoivat kehittäjiä nopeasti tunnistamaan ja korjaamaan ongelmia. Tämä paransi koodin laatua ja vähensi virheiden määrää tuotantoympäristössä.

Tutkimus on hyödyllistä ohjelmistokehittäjille, testaajille ja projektipäälliköille, jotka työskentelevät monimutkaisten web-sovellusten parissa. Se tarjoaa arvokasta tietoa ja käytännön ohjeita automatisoidun testauksen käyttöönotosta ja integroinnista. Tutkimus voi olla hyödyllinen myös muille Wärtsilän projekteille ja työkaluille, joissa voidaan hyödyntää samoja testausmenetelmiä ja työkaluja.

6.2 Haasteet ja ratkaisut

Vaikka Playwrightin asentaminen osaksi FeasibilityTools-työkalua oli helppoa hyvän dokumentaation ansiosta, suurimmat haasteet liittyivät Playwrightin integroimiseen olemassa olevaan CI/CD-putkeen. Dokumentaatiosta eikä Googlesta löytynyt paljon tietoa siitä, miten Bamboo

ja Playwright saadaan toimimaan yhdessä. Tämä johti siihen, että ongelman ratkaisua piti pohtia paljon itse ja kollegoiden kanssa. Haasteiden ratkaisemiseksi oli tärkeää hyödyntää tiimin osaamista ja kokemusta, sekä testata erilaisia lähestymistapoja, kunnes löydettiin toimiva ratkaisu.

E2E-testauksen suunnittelussa elementtien paikantaminen on toisinaan haastavaa. Web-sivujen tilan muutokset voivat aiheuttaa sen, että elementti ei löydy odotetusti ja testi epäonnistuu. Lisäksi local- ja dev-ympäristöissä käytetty erilainen data voi johtaa vastaaviin ongelmiin.

Vaatimukset Playwrightin asennusta ja integrointia kohtaan täyttyi ja täten tutkimuksen näkökulma onnistui. E2E -testejä on vielä niukasti ja niiden toteutus on vielä työn alla. Jatkossa testeissä keskitytään, että testit olisivat mahdollisimman helppo ylläpitää.

6.3 Jatkokehitys

Jatkossa olisi hyödyllistä kehittää lisää dokumentaatiota ja ohjeita Playwrightin ja Bamboo CI/CD-järjestelmän yhteiskäytöstä Confluenceen. On ollut myös puhetta, että voisi laajentaa Playwrightin käyttöä muihin Wärtsilän projekteihin ja työkaluihin.

Tulevaisuudessa keskitymme E2E-testien suunnittelussa erityisesti niihin elementteihin, jotka voivat muuttua. Tämä ongelma voidaan ratkaista lisäämällä elementeille yksilölliset id-tunnisteet. Näin testit voivat paikantaa tarkasti kyseisen elementin ja jos tulee koodimuutoksia, niin sen ei pitäisi vaikuttaa testituloksiin.

LÄHTEET

Atlassian. (2024). Bamboo - Continuous integration, deployment, and release management. Viitattu 13.11.2024. https://www.atlassian.com/software/bamboo

Björkman, M. (2024). SOFTWARE TEST AUTOMATION: Implementation of End-to-End testing in web application. Viitattu 04.11.2024. https://www.diva-portal.org/smash/get/diva2:1850362/FULLTEXT01.pdf

Docker. (2024). Docker overview. Viitattu 13.11.2024. https://docs.docker.com/get-started/

Fewster, M. & Graham, D. (1999). Software Test Automation: Effective Use of Test Execution Tools. Addison-Wesley. Viitattu 13.2.2025. https://books.google.fi/books/about/Software_Test_Automation.html?id=in7gTCu5SE8C&redir_esc=y

Guru99. (2024). CI/CD Pipeline: Learn with Example. Viitattu 12.11.2024. https://www.guru99.com/ci-cd-pipeline.html

Hetzel, W. C. (1988). The Complete Guide to Software Testing (2nd ed.). Wiley. Viitattu 13.2.2025. https://pdfcoffee.com/bill-hetzel-the-complete-guide-to-software-testibookseeorg1-pdf-free.html

Jorgensen, P. C. (2016). Software Testing: A Craftsman’s Approach (4th ed.). CRC Press. Viitattu 13.2.2025. https://malenezi.github.io/malenezi/SE401/Books/Software-Testing-A-Craftsman-s-Approach-Fourth-Edition-Paul-C-Jorgensen.pdf

Katalon. (2024). What is End-to-End Testing? Definition, Tools, Best Practices? Viitattu 25.11.2024. https://katalon.com/resources-center/blog/end-to-end-e2e-testing

Koskela, L. (2013). Effective Unit Testing: A Guide for Java Developers. Manning Publications. Viitattu 01.11.2024. https://www.academia.edu/43459211/M_A_N_N_I_N_G_A_guide_for_Java_developers_LASSE_KOSKELA

Memon, A. (2016). Advances in Computers. Viitattu 24.11.2024. https://www.sciencedirect.com/science/article/abs/pii/S0065245815000686

Microsoft. (2023). What is DevOps? Viitattu 12.11.2024. https://learn.microsoft.com/en-us/devops/what-is-devops

Myers, G. (2012). The Art of Software Testing (3rd ed.). Wiley. Viitattu 01.11.2024. https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf

Narendar, K. (2022). Integrating Continuous Integration / Continuous Deployment (CI/CD) with Test Automation: Enhancing Software Development Efficiency. Viitattu 13.11.2024. https://www.academia.edu/121716452/Integrating_Continuous_Integration_Continuous_Deployment_CI_CD_with_Test_Automation_Enhancing_Software_Development_Efficiency?nav_from=b941bf6a-abe3-4e2c-b7bc-dec130579564

Pyhäjärvi, M. (2020). Test Automation Process Improvement in a DevOps Team: Experience Report. Viitattu 01.11.2024. https://www.academia.edu/49495368/Test_Automation_Process_Improvement_in_a_DevOps_Team_Experience_Report

React. (2024). Viitattu 13.11.2024. https://en.wikipedia.org/wiki/React_(software)

Saltzer, J.H., Reed, D.P., & Clark, D.D. (1984). End-to-end arguments in system design. ACM Transactions on Computer Systems. Viitattu 04.11.2024. https://dl.acm.org/doi/pdf/10.1145/357401.357402

Sommerville, I. (2015). Software Engineering. Pearson. Viitattu 13.2.2025. https://dn790001.ca.archive.org/0/items/bme-vik-kony-vek/Software%20Engineering%20-%20Ian%20Sommerville.pdf